THE ANALYSIS AND EXTENSION OF AN EXISTING DATA LINK PROTOCOL

by

ALAN LEON VARNEY

B. S., Kansas State University, 1970

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:

Richard A. McBride

Major Professor

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Communication protocols[1] are widely used in computer-to-computer information transmission for routing, error detection and flow control. Many protocols (computer and non-computer) have been proposed over the past several decades. The earliest technical protocols dealt with telegraph-like communication [Fah74]; early hardware was typically used to send "messages" to a receiver via a series of shocks, since the light bulb and electromagnetic relay were invented years after these original protocols. Later techniques used magnetically-operated levers and signals to send letters and numerals using some ingenious and compact methods[2]. Protocols proposed over the last several years have steadily increased in complexity to better use the available communication media and to handle an increasing volume of information.

This work reports on an existing basic communications protocol and then extends that protocol with previously published mechanisms [Tan81, Bla83, Dav83]. The analysis of the extended protocol and it's implementation in a concurrent language environment (Concurrent C) are also described. The environment also provides an interface for testing this and other protocol implementations within a limited architecture. Several scenarios are presented to show that the enhanced protocol recovers properly from "seeded" errors.

The work has application in the areas of interprocessor communication and data transmission. Since the protocol provides "reliable" node-to-node transmission of data, it is possible to use it as part of a reliable terminal-to-mainframe communication protocol. The

---

1. Protocols are the rules and conventions used to communicate between entities or processes.
2. One such protocol's summary ended "With such an apparatus no errors could possibly occur, for everything went like clockwork." [Amy38] Others believed "...success...would likewise depend no an apparatus liable to an infinite number of accidents, scarce in the power of human foresight to guard against." [Gam97] Protocol designers should keep the latter comment in mind.

concurrent-language test-bed offers a mechanism to test new protocol implementations at the "user" level without the need to create a new test environment.

Chapter 2 summarizes a portion of the abundant literature on communication problems and protocol analysis and design. The basic protocol mentioned above is also described in some detail. In Chapter 3, some possible extensions to the basic protocol are presented. Some of these extensions are then integrated into the basic protocol, and an analysis of the enhanced protocol is presented. Chapter 4 offers a concurrent language implementation of the enhanced protocol together with a test generator/monitor. The conclusions in Chapter 5 also present some areas for further investigation and discuss other implementation possibilities.

2. EXISTING LITERATURE

Communication protocols and their analysis have been a topic of a large volume of literature, both in book and journal form. Unfortunately, much of the design and implementation of protocols was originally performed by equipment manufacturers, with the usual results: Each manufacturer used a private protocol, incompatible with all others. Eventually, the major manufacturers' (primarily IBM and Digital Equipment) protocols became *de facto* standards. The literature reflected this proliferation of standards by using inconsistent and ambiguous terminology. Attempting to define a "common ground" for discussion, several standards organizations together created an architectural model for communication processes. The result is called the Reference Model of Open Systems Interconnection (hereafter called the OSI Model), sponsored by the International Standards Organization. [Zim80]

2.1 OSI Model

To place the protocols mentioned below into their proper perspective relative to the overall process of communication, the OSI Model will first be summarized. The OSI Model architecture partitions communication into seven layers. Within an entity (also called a *node*), each numbered layer "interfaces" functionally only with its adjacent layers by providing services to the layer above and using services of the layer below. Communication between entities logically takes place only between like-numbered layers using a protocol for that layer. These protocols operate using the services provided by the next lower layer. The defined layers and their numbers are:

Application Layer (7) - provides support directly to the Application Processes executing across the various communicating entities (for example, process synchronization).

Presentation Layer (6) - provides data transformations between Application Processes (e.g., compaction/expansion).

Session Layer (5) - provides the "dialogue" connection between pairs of Application Processes (via Level 6) and the capability to establish the connection.

Transport Layer (4) - provides reliable end-to-end transmission of messages between communicating processes.

Network Layer (3) - provides the means for sending messages between any pair of entities within the network, typically providing routing and flow control on an end-to-end basis.

Link Layer (2) - provides the means to reliably exchange sequences of messages between directly-connected entities.

Physical Layer (1) - provides the electrical and mechanical connection between entities and services to the Link layer allowing message transmission.

Typically, a message from a higher-numbered layer is transported by prefixing a layer "header" to the message and sending the message to the next lower layer. Message reception operates in the reverse order; receiving messages from the lower layer, removing and processing the layer "header" and delivering the remaining portion of the message to the next higher layer.

2.2 Link Layer

This paper will be primarily concerned with protocols at the Link layer (Layer 3). Thus issues such as physical media (Layer 1) and routing (Layer 3) will not be covered. To provide services to the Network layer as mentioned above, the Link layer must be able to:

1. identify or address the required physical unit for transmission,
2. maintain a logical sequence of messages for each logical data link,
3. handle errors in message ordering and transmission and
4. control the flow of messages on the physical connection.

These capabilities are provided by "framing" the messages into blocks that have a structure determined by the protocol. The Link layer sends and receives these *frames* [3] through

services provided by the Physical layer. The Physical layer's primary service is the transmission of bit or character streams over a logical link.

The Link layer of a particular network topology is influenced by that topology, even though routing through the network is not a function of this layer. This is because connections other than *node-to-node* (i.e., between directly connected entities) are supported by the OSI Model. These other connections are termed *multipoint*; this type of Link layer connection is not covered in this paper. Node-to-node connections at the Link layer are "logical links" in that one real transmission path may support several logically distinct connections between the same pair of nodes. The interface between the Link layer and the Physical layer would, in this case, require a logical link identification on every exchange of information to maintain the logical link association. Within the node-to-node type of connection, two methods of "framing" the messages are commonly used. One of these is known as *character-oriented;* the other is called *bit-oriented.*

The bit-oriented protocols are newer and have several advantages over the character-oriented protocols. The basic unit in the frame is a bit; certain bit sequences are used to detect the beginning and ends of frames. The frame can typically contain any number of bits. These protocols are inherently transparent to character set size and codes through the use of a technique called "bit-stuffing." (This replaces certain bit sequences with other sequences that have extra zero bits within them, thus preventing the framing bit sequences from appearing within the unit sent to the Physical layer.) The bit-manipulation required (monitoring for

3. The term *frame* is used in this chapter to refer to data units exchanged between nodes at the Link layer. The OSI Model's corresponding term is "physical-layer-service-data-unit," but one suspects this is somewhat shortened in informal conversation.

special bit sequences and shifting to add or delete the extra zero bits) almost forces the Link layer (or a portion of it) into hardware; the overhead on general-purpose computers would overcome most of the advantages. Some examples of bit-oriented protocols are:

HDLC (High Level Data Link Control) - part of ISO X.25,
ADCCP (Advanced Data Communication Control Protocol) - ANSI,
SDLC (Synchronous Data Link Control) - part of SNA from IBM,
BDLC (Burroughs Data Link Control) - Burroughs and
UDLC (Univac Data Link Control) - Univac.

2.3 Character-oriented protocols

Character-oriented protocols are based on characters as the unit of transmission. Typically, a small set of seldom-occurring characters are chosen as "control" characters. The occurrence of these characters in the input is used to delimit and define the frame sent between nodes at the Link layer. The following functions are basic [Con80] to character-oriented Link layer protocols:

1. *Framing* places a structure on the character stream received from the Physical layer by identifying the beginning and end of frames.
2. *Error Detection* provides a means to determine if frames were altered in transmission. It also provides for detection of out-of-order frames.
3. *Recovery* controls the actions taken to recover from errors and lack of response from the connecting node.
4. *Flow Control* handles the rate of character flow between nodes.
5. *Link Control* allows the connection between nodes to be established and terminated.

Each function will be described in some detail below, except for *Flow Control*, which is typically handled by ignoring frames that the node is not prepared to receive. Also, a *Transparency* function is provided by many character-oriented Link layer protocols; this allows "control" character bit sequences to be sent as data without being considered "control" characters by the *Framing* function.

Framing is accomplished by using a particular "control" character to identify the beginning of a frame. The end of the frame is identified either by a different "control" character or by a character count contained within the frame. (The latter method is usually called a *byte-count character-oriented* protocol.) The *Framing* function scans the incoming characters from the Physical layer for the beginning-of-frame character. Everything after that until an end-of-frame character (or the number of characters in the character-count) is considered a frame.

Error Detection of transmission-induced alterations to the frame is commonly performed by *checksums*. A checksum is the result of a computation over the characters in the frame. The checksum is sent with the frame; on receipt, the checksum computation is performed again and compared to the transmitted checksum. The frame is considered "damaged" if the two checksums are not equal[4]. The most commonly known checksum methods are:

ARPANET Internet – "...the 16-bit one's complement of the one's complement sum of the 16-bit of all 16-bit words..." [Pos81],

CRC-16 – the remainder after dividing the frame, treated as a single binary number, by the binary number 1100000000000101, and

CCITT V.41 – the remainder after dividing the frame, treated as a single binary number, by the binary number 1000100000100001.

The latter two methods (known as Cyclic Redundancy Codes) can be shown [Tan81] to detect any odd number of incorrect bits, any two bits incorrect, any error contained within a 16 bit sequence and 99.998% of any other random bit errors.

Error Detection of out-of-order frames requires a sequence number field within the frame. Each node has a variable that determines the next frame sequence number to expect from a

4. It is possible for the Physical layer to damage a frame in such a way that the checksums will still match. The checksum method should make that possibility very remote by detecting well the most likely errors on the particular type of physical link(s) used in the network.

connecting node. If a received (undamaged) frame does not contain the expected sequence number, the frame is considered *out-of-sequence*. If the sequence number is correct, an *acknowledgement* (ACK) is sent back to the transmitting node to indicate successful reception of an in-sequence frame. The transmitting node must retain all transmitted messages until informed by the receiver that transmission of the frame was successful.

The *Recovery* mechanism is a major part of a data link protocol. There are many types of recovery possible. The simplest mechanism sends a single frame and waits a (predetermined) length of time for the receiver to send back an ACK. Failure to receive the ACK will result in a "timeout" of the waiting transmitter and retransmission (and another wait) will occur. This continues until the frame is accepted. The receiver must continue to respond with an ACK even if the frame has been successfully received, to insure that its own ACK wasn't lost. This mechanism is called the stop-and-wait ARQ (Automatic Repeat Request) or PAR (Positive Acknowledgement with Retransmission) method.

This can be enhanced somewhat with a *negative acknowledgement* (NACK) sent by the receiver to inform the transmitting node that a "damaged" frame has arrived. This would result in immediate retransmission rather than waiting for timeout. Timing is still required to detect complete loss of the frame, ACK or NACK.

Further improvements are possible only if *pipelining* of the transmitter's frames is allowed; this allows a small number of frames to all be transmitted without waiting for an ACK. The number of sent but unacknowledged frames allowed is called the *send window*. If the receiver still insists that frames will only be accepted in order, the receiver is said to have a *receive window* of size 1. When the receiver sends a NACK, it will contain the sequence number that was last received correctly[5]. This allows the transmitter to determine from

which frame to begin retransmission. This retransmission strategy is called "Go−Back−N." Each frame in the send window must be timed separately. Timeout of a frame will result in the same actions as receipt of a NACK indicating the frame was lost (i.e., the frame and all following frames will be retransmitted).

If the receiver can also accept some frames out-of-sequence (and hold them until predecessors arrive), the receive window size determines the number of out-of-sequence frames that can be accepted. However, the receive window cannot be larger than N/2 where N is the number of unique sequence numbers. With such a receive window, an improved retransmission strategy called "Selective Repeat" may be used. The transmitter, on receipt of a NACK+sequence number, can retransmit only the next frame in the sequence. If that frame was the only missing one, the receiver will ACK with a sequence number that acknowledges the retransmitted frame and the frames that the receiver had previously accepted out of order. Timeout of a frame would result in retransmission of only that frame.

Another enhancement possible with pipelining is called *piggybacking*. This allows an ACKed sequence number to be placed in data frames that are being transmitted back to the original sending node, thus avoiding the transmission of a separate ACK frame. Since the ACKed sequence number typically is only a few bits in size, there is little frame overhead involved in this enhancement. However, to fully exploit piggybacking, the receiving node must delay transmitting ACKs for a time to allow transmission requests in the reverse direction to arrive. If there is no reverse traffic, then an ACK must be transmitted or the protocol will eventually "lockup" (i.e., stop sending new frames). Delaying for a period of

5. Some protocols indicate the sequence number *expected next*, instead of the *last received* number.

two or three ACKs has been shown [Lai82] to result in a significant amount of piggyback utilization, if the send window is of size 4 or more. If the number of "accumulated" ACKs is allowed to equal the window size, lockup will occur until some reverse traffic appears.

Link Control provides the mechanism needed to establish a connection over a link or to disconnect such a connection. The protocol attempts to keep both ends of a link either "connected" or "disconnected." In the disconnected state, no Network layer requests (except for *connection*) are accepted; most frames from the link receive a "I'm disconnected" response. Requests for *disconnect* while in the connected state may be honored immediately or may be deferred until there are no further unacknowledged frames at either end of the link. *Link Control* must also handle the case of a request for *connect* or *disconnect* when the state is already connected or disconnected, respectively.

Transparency is inherent in the byte-count protocols, but requires an "escape" mechanism in other character-oriented protocols. The escape mechanism operates by replacing "control" characters in the frame with a sequence of characters before transmission. The character sequences are replaced with the original characters when the frame is received. This is called "character−stuffing," in analogy to the "bit−stuffing" of bit-oriented protocols.

2.4 Examples of character-oriented protocols

Most of the earliest computer-communication protocols were character-oriented. The early teletype networks used 5 or 6 bit Baudot codes with embedded control characters to control the receiving devices. A list of major computer-to-computer protocols includes:

BISYNC (IBM's Binary Synchronous Communication protocol) is widely used, particularly in a multi-point configuration to handle groups of remote terminals. The node-to-node version operates over half-duplex lines.

X3.28 (ANSI's protocol) standardizes on 10 ASCII control characters for link control and specifies several variations of a protocol for different

configurations and applications.

IMP-to-IMP (ARPANET's sub-network protocol) has operated for 15 years providing service to well over 100 host computers. It operates 8 logical links over each physical link using a stop-and-wait protocol. NACK is not used, but the piggybacked acknowledgement field does contain the last-correctly-received frame number.

DDCMP (Digital Equipment's Digital Data Communications Message Protocol) uses the byte-count mechanism to achieve data transparency.

Other protocols that are less well-known usually use only minor modifications of the previous protocols. However, some notable exceptions to this have been implemented. The MININET [Ner84] local area network uses continuous transmission of short (32 bit) fixed-length frames to achieve a "lock-step" form of operation. By establishing a count of the number of frame periods required to send and receive initialization frames, each node can determine the number of frames always in transit between itself and another connecting node. No ACK or NACK control frames exist. Receipt of a damaged frame is indicated to the transmitter by having the receiver backing up its own frame sequence by the number of transit frames + 1, and retransmitting frames from that sequence number forward. The transmitter, on receipt of an out-of-sequence frame, backs up an amount determined by the out-of-sequence frame number and the frame number it expected to receive.

The French NADIR satellite link protocol [Ner84] uses explicit ACKs for every undamaged frame received, regardless of order. However, the ACK contains two sequence numbers and acknowledges receipt of all frames between those two numbers. Thus the transmitter can determine which frames, if any, have been lost or damaged and can thus retransmit them immediately. There is little need for a NACK with this method. In heavy traffic, the ACK for the following frame will indicate a missing frame. In light traffic, a timeout will eventually cause the frame to be retransmitted; in light traffic the time saved by a NACK

would have little impact.

2.5 Comer's Protocol – Overview

Comer, in an excellent book describing both a layered approach to and implementation of an operating system design [Com84], presented a simple, effective data link communication protocol. The protocol was presented in two sub-layers called Data Link level and Frame level[6]. Comer's *levels* together approximate the OSI Model's Link layer. The Data Link level is also similar to the Media Access Control sub-layer of IEEE Standard 802 [IEE84] with the Frame level acting as the IEEE802 Link Level sub-layer. However, Comer's Frame level protocol contains somewhat different capabilities than the IEEE802 Link Level sub-layer, since the latter has a virtual connection service available to higher-numbered layers. Comer's Frame level protocol does, however, offer reliable transport of individual data packets[7] with no lost or out-of-order packets. (Detection of duplicates, if needed, must be performed at a higher layer.)

Comer's protocol is based on a particular processor inter-connection scheme called a unidirectional ring-shaped network. Each processor (or *machine*) in the network has an asynchronous, full-duplex bit-serial connection to two other processors[8]. The processors and connections together form a ring over which "frames" of character data are transmitted. One connection on each machine is considered the input port and is connected to the output port

6. The phrase "Data Link level" will be used to refer to Comer's "sub-layer" and it's extension in the next chapter. The phrase "Link layer" will be used to refer to that layer in the OSI Model.

7. A "packet" is the unit of data exchanged at the Network layer. It is also, in this paper, used to describe the data the user wishes to send between two process existing on different physical machines. In effect, this lumps all data transmission requests to the Link layer from higher layers into the term "packet," regardless of the source of the data.

8. It is possible for one machine to be in more than one ring, thus allowing it to be used as a "gateway" for communication between the rings. That capability does not affect the protocol used over a single link, since the routing between networks is a function of the Network layer.

of a "neighboring" machine. The other connection is the output port and connects to the input port of a (usually) different machine. The flow of data messages between machines is always in one direction, and response characters (for ACK, etc.) are sent only in the reverse direction.

The general format of a frame in Comer's protocol is shown in Figure 1 along with the general interconnection of processes.

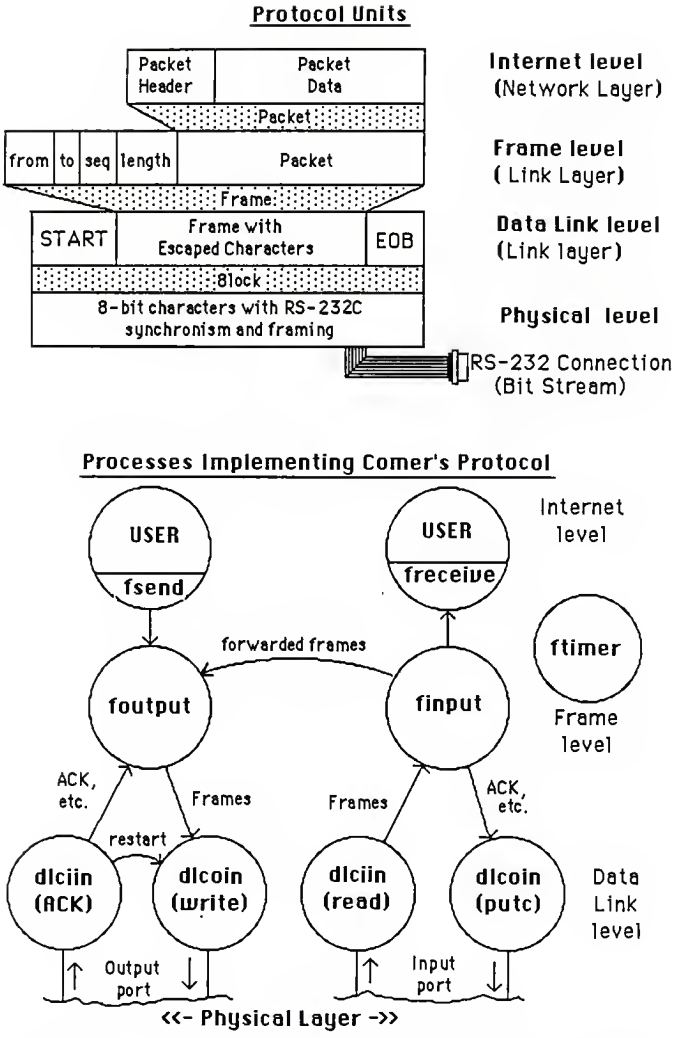


Figure 1. Comer's Protocol - Frame and Process Overview

Application processes request the transmission of packets to other machines on the ring. These packets are placed into frames and transmitted via the output port. Transmitted frames are received at the input port of the "next" machine on the ring, validated and examined to determine their source and destination. An ACK is sent back to the transmitting machine to indicate the frame was successfully received. Frames destined for the receiving processor are passed "up" a layer to a process that distributes the frames to various other application processes. Frames not destined for the receiving processor are usually forwarded via the output port to the next processor. However, if *no* processor "claims" a frame, it will eventually make its way around the ring and return to the originating processor; such frames with a source equal to the receiving processor do not continue to be forwarded. In this case, an error message is printed to the console. Please note that this overview does not describe Link layer error detection or recovery mechanisms.

2.6 Comer's Protocol – Data Link level

There are two processes at the Data Link level associated with each port. However, input port processing differs greatly from output port processing. The difference in processing is a result of requests from the Frame level, not in an internal knowledge of the type of port being served.

Each process is a classical *interrupt-driven device handler*. Comer, however, allows these to be viewed as a type of process that "waits" on interrupts when asked to do so by a Frame level request. The input interrupt handler (*dlciin*) waits on incoming characters, receiving control when a character has arrived. (*Dlcoin*), the output interrupt handler, waits for outgoing characters to be transmitted, receiving control when it's time to send another character. While no appropriate Frame level request is active, any interrupt is ignored.

Data Link processing at the output port is initiated by the Frame level via a *write* request to *dlcoin*. Prior to the *write* request, a process must be registered with the ACK handler portion of the associated *dlciin* process to receive the ACKs, etc. sent to the output port by the receiving machine. The (*dlcoin-write*) process transmits a frame after "framing" its input data with the START character at the beginning and the EOB character at the end. In order to prevent these and other protocol-related "control" characters from appearing in the packet portion of the transmitted frame, "character-stuffing" is performed. "Control" characters in the packet are replaced by an ESC (escape) character followed by the "control" character modified by zeroing a particular bit. (All "control" characters have the particular bit set to one.)

The *dlciin-ACK* process will send any characters received on the output port to the previously registered ACK-receiving process, with one exception. If the received character is RESTART, *dlciin-ACK* checks the state of the associated *dlcoin-write* process by examining a shared variable. If the state is "INIT" (implying *dlcoin-write* is not currently transmitting a frame), the RESTART is sent to the registered process, as before. If, however, the state is anything other than "INIT," *dlciin-ACK* will force *dlcoin-write*'s state to "RESTART." The "RESTART" state informs *dlcoin-write* to stop transmitting the current frame and start retransmitting from the beginning. The RESTART character thus serves as a quick method of informing *dlcoin-write* that the frame currently being transmitted has already been rejected, and continued transmission is useless.

Data Link processing at the input port is also initiated at the Frame level, via a *read* request to *dlciin*. The *dlciin-read* process examines incoming characters, looking for the START character. It then stores the packet from the incoming frame, after "unstuffing" the two-

character ESC sequences by setting the previously zeroed bit in the character following ESC. The frame ends when the input buffer fills or EOB is received. During this processing, some character-related transmission faults can be detected. These are:

- a hardware-detected error (carrier loss, framing error, data overrun, etc.) and
- a character with the special "zeroed bit" already set following an ESC.

Any such fault causes *dlciin-read* to send a RESTART character back to the transmitting machine. The *dlciin-ACK* process would receive the RESTART and either initiate retransmission of the frame (as described previously) or notify the Frame level registered ACK process. The bit representation of RESTART has been chosen to not overlap ACK, NACK or SACK[9] responses; the output process treats anything other than ACK or SACK as a NACK, so the net result of a RESTART is retransmission of the frame.

The Frame level sends ACK, NACK or SACK responses to the transmitting machine using the *putc* request. The *dlcoin-putc* process just transmits the requested single-character response.

2.7 Comer's Protocol – Frame level

There are three processes at the Frame level. The output Frame level process (*foutput*) accepts data for transmission from application processes, adds destination and sequencing information, sends the packet to the output port via (*dlcoin-write*), and retains a copy until successfully acknowledged by the next machine's input Frame level process (*finput*). Unacknowledged frames are also timed for retransmission. The *finput* process receives

9. SACK is a type of NACK that will also cause the expected sequence numbers at the sender and receiver to be reset. This will be described later.

frames from the input port via (*dlciin-read*), validates them and either sends the frame's data to the appropriate application process or forwards the frame to the next machine (via another *foutput* process). It also sends an ACK back to the transmitter via *dlcoin-putc*. A third process (*ftimer*) provides the *foutput* process with the timeout indication needed to force retransmission of an unacknowledged frame. The *ftimer* process sleeps unless such a frame exists.

To detect invalid frames, *finput* first verifies that the number of characters in the frame matches the length field within the frame. Missing or out-of-order frames are detected by assigning a sequence number to each frame when it is received from an application process. Comer's protocol uses a 3-bit sequence number in each frame to detect missing or out-of-order frames; each new frame is assigned an incremented sequence number (modulo 8). After sending a frame (via *dlcoin-write*), *foutput* waits for *finput* (in the receiving machine) to acknowledge the frame; this is commonly known as a stop-and-wait ARQ (Automatic Repeat Request) technique. Failure to receive an acknowledgement will be detected by the *ftimer* process and the unacknowledged frame will be retransmitted.

When *finput* receives a frame, the sequence number is compared to the expected sequence number. If they match, an ACK "character" with the "next expected" sequence number embedded is sent back to the transmitter. If the ACK is successfully received by the transmitting *foutput* process and the embedded sequence number matches the next expected sequence number, the unacknowledged frame is removed from *foutput*'s buffers and the *ftimer* process is put to sleep. If the received frame's sequence number doesn't match, a NACK "character" (with the current expected sequence number embedded) is sent to the transmitter. The transmitting *foutput* process will retransmit the unacknowledged frame

(without examining the NACK's expected sequence number) and wait for a response (ACK or NACK or SACK).

The transmitting and receiving ends of the link could get out of sync with regard to sequence numbers. If *finput*'s expected frame number is 4 and the message sitting in *foutput*'s buffer is 3 (or anything other than 4), then sending NACKs back to the *foutput* process will never (without some improbable bit mutilation) result in a message with a sequence number of 4 being retransmitted. To "solve" this problem, the *finput* process will send a SACK (in place of a NACK) if sequence number mismatches cause more than two NACKs on the same incoming message.

SACK informs the *foutput* process to reset it's current sequence number back to 0; *finput* also resets its expected sequence number. Sequence number 0 will now be used to re-transmit the unacknowledged message and that number will be acceptable to the *finput* process. Note that the above scenario is not that improbable; if the ACK for frame 3 was lost, *foutput* will time-out and retransmit frame 3. However, *finput* will be willing to accept frame 4 only and will respond with a NACK. If this occurred, *finput* would eventually accept the data in frame 3 twice[10], once as frame 3 (normally) and again as the new frame 0 (after NACKing three times).

Comer's protocol allows a single message to be transmitted to all machines on the ring network; this is known as a broadcast message. One particular value (zero) in the destination field of the frame header implies that the frame should both be passed up to the

10. Comer states that detection of duplicate messages must be at a layer above the Frame level. The detection mechanism must *not* rely on the Frame sequence numbers.

application's distribution process and forwarded to the next machine. When the frame has been forwarded completely around the ring, the originator of the broadcast message stops forwarding the frame.

Aside from the duplicate message problem and the lack of any rigorous bit mutilation detection, Comer's data link protocol appears to perform well. However, there are several extensions that would enhance it's efficiency and robustness, as well as adapt it for use in a non-ring network. All this is the subject of the following chapter.

3. PROTOCOL DESIGN

There are several problems with Comer's protocol (see below) which make it unsuitable for many applications. This chapter will describe possible extensions to the protocol and choose a subset of those extensions for further discussion. The resulting extended protocol will be described in detail, both in text and in modified state transition diagrams. Several scenarios will then be presented that show its operation under normal and failure conditions.

Two types of extensions to Comer's protocol are possible; those that add new capabilities (features) to improve efficiency, control, and flexibility (allowing alternate applications and architectures) and those that repair problems in the existing protocol. Note that while no formal verification of Comer's protocol was performed, examination of both the English description and the program code revealed several problems with the protocol. These include:

- no checksum over any portion of the frame,
- the possibility of passing duplicate packets to the Network layer,
- failure to detect excessively long frames and
- failure to detect a non-working link.

3.1 Possible Extensions

Any *checksum* mechanism could improve the ability of the protocol to withstand transmission errors. It is unusual to find any protocol that does not use at least a simple exclusive-or or sum-of-the-characters checksum.

Duplicate packets may be prevented by enhancing the NACK mechanism and removing the SACK acknowledgement. The detection of a *long frame* is a simple change; receiving a non-EOB character after the frame buffer is full would force the frame to be treated as

invalid. (Comer's implementation *assumed* an EOB would follow and passed the frame up to the Frame level.)

Another extension could be the addition of a *heartbeat* frame; it is periodically transmitted in the absence of other outgoing frames. The receiving process would complain if no incoming frames were detected over several *heartbeat* intervals. This would detect link failures (or lockup at the other end) even in the absence of outgoing traffic.

One capability that could add flexibility is *operation on non-ring architectures*. This would require that frames be passed to a "routing" process at the Network layer. This process would deliver frames after examining their destination and allow the output link for that destination to change over time.

A further extension could allow *full-duplex transmission of frames* over a link. Major changes to the single-character ACK/NACK/SACK mechanism would be required. Also, the Data Link input process must allow frames and acknowledgements to travel in both directions on a link.

Further flexibility could be provided by offering *virtual circuit* facilities at the Link layer, as proposed in [IEE84]. This would require (at least) the addition of another sublayer within the Link layer.

A related extension could be the provision of *datagram* facilities. This would allow a packet to be transmitted without retransmission or sequencing control. Errors in such packets would force them to be discarded without indication to either the originating or transmitting node.

A final extension for flexibility could add *unbalanced mode* operation to the protocol, in the spirit of HDLC. Several versions of Link Control (master, slave, etc.) would have to be

added, as well as a means for nodes to mutually agree on a version to use over a period of time. One could even imagine adding the ability to operate in a *multi-point* configuration (i.e., a master and several slave nodes connected to the same link), although this would require changes to the (assumed) RS-232C Physical layer.

A control extension could provide the ability to *stop (and start)* traffic over a link without loss of frames. This would allow a user-level process to request that the link be logically "disconnected" after ensuring that all preceding frames had been transmitted. All further requests for packet transfer from the user lever would be blocked, but a request to reconnect the link would be allowed from either end of the link. Such a mechanism must ensure that neither end of a link is "disconnected" when the other end is "connected."

Another control extension could allow *direct control and monitoring* of the Link layer via user "Link control" messages. This would allow a user-level process to retrieve the status of the Link layer of a given link from any node, providing a method for collecting statistics on performance from a central location. It could also allow a remote node to "connect" or "disconnect" the link, as well as simulate packet input to the Link layer. If a "download protocol" capability also existed, it would be possible to load new protocols (of any layer) into the node from remote locations. Collectively, these capabilities would allow unattended operation of some nodes of the network.

Pipelining of frames and *piggybacking* of acknowledgements are obvious efficiency extensions. With pipelining comes the choice of various possible retransmission strategies such as "Go-Back-N" or "selective repeat." Pipelining would make the SACK resynchronization mechanism even worse since it could cause several duplicate frames to be passed to the Network layer. The effectiveness of Comer's RESTART mechanism could be

affected by the particular implementation chosen[11]. Piggybacking, to be effective, would require that a few ACKs be postponed for a short period waiting for packets in the reverse direction.

3.2 Choice of Extensions

The following set of extensions has been chosen for use in the remainder of this paper. Where necessary, the reason for choosing that extension is discussed.

- Checksums are added to all frames, via the CRC-16 technique.
- Duplicate frames are never passed to the Network or user levels. The SACK acknowledgement is dropped. Both ACK and NACK will carry the sequence number indicating the last frame received in sequence.
- Excessively long frames are detected and treated as "damaged" frames. It is unlikely that such frames would survive the checksum mechanism (approximately one chance in 2^{16}), but this extension is almost trivial.
- The *heartbeat* capability is added.
- Various network architectures are supported, although this is achieved primarily by inventing a routing process in the Network layer. The method by which the routing process performs its task is beyond the scope of this paper, however[12].
- Full-duplex transmission of packets and acknowledgements is added. The acknowledgement and control "characters" from Comer must be converted into control frames and distinguished from frames bearing packets.
- A "connect" and "disconnect" control facility is provided. The link is thus in either a "running" or "disconnected" state. User *fsend* requests and forwarded packets are denied in the "disconnected" state and the *heartbeat* mechanism is disabled. Incoming frames other than those associated with link control are discarded, with an AMDISC control frame being sent back to the transmitter.
- Pipelining is added to allow more efficient use of the physical link. The "Go–Back–N" retransmission strategy is chosen for both its effectiveness and

---

11. RESTART causes the current Data Link output transmission to abort. In Comer's protocol, this allowed retransmission of the current frame to begin immediately. With pipelining, the frame containing the error may not be the one being transmitted. If the retransmission strategy allows frames to arrive out of order, as in "selective repeat," then little would be gained in aborting transmission of a frame that was probably on its way to being accepted.

12. While this extension removes a capability from the original protocol (explicit forwarding of frames), note that Comer's routing at the Link layer was possible only because of the architecture. However, routing "off the network" via gateways was mentioned as a Network layer task in [Com84]. His total description of the routing process was that it "... computes a new route for the packet ..." I am thus following (at least here) in Comer's footsteps.

simplicity. Comer's RESTART mechanism will be only partially used. A means will be provided to abort any current frame transmission when frame retransmission is needed. Retransmission will be initiated at the Frame level. No use will be made of a RESTART character.

- Piggybacking of positive acknowledgements is added.

In choosing the particular extensions to make to Comer's protocol, the goal was to provide a non-trivial set of extensions without attempting to do everything mentioned above. The extensions listed above are certainly non-trivial. They entail changes to all areas of Comer's protocol, and also alter the interfaces to the adjacent layers.

3.3 Interfaces to Other Layers

Comer's Network layer could make only two requests of the Link layer for a given link, *fsend* and *freceive*. The extended protocol recognizes a third request, *fctrl*, to either establish the link connection or to disconnect the link. For a given link, a single process is still identified to receive all incoming messages (packets destined for this node). This process is responsible for distributing the messages to the proper applications.

The extensions require the addition of a routing process to the Network layer. Any accepted incoming frames are passed to this process. The Network layer sends outgoing packets to the Link layer for transmission; these have a priority indication so that, for example, forwarded packets may be transmitted before *fsend* packets that have not yet been transmitted.

No major changes in the interface to the Physical layer were required by the extensions. The full-duplex transmission capability of the layer is now being fully utilized. Both input and output continue to be viewed as a stream of characters. The single character ACK and NACK can no longer be used, since they could be confused with extraneous characters between frames; they also can benefit from the use of a checksum.

3.4 Protocol Overview

To simplify the presentation of the overall data link protocol, it has been divided into four sublayers [Boc79]. These correspond (almost) to the basic functions of character-oriented Link layer protocols listed in the previous chapter. These sublayers are (from lowest level to highest):

- Framing and Transparency - determining the actual characters sent over the physical link,
- Error Detection - validating the incoming frame,
- Data Transfer - controlling the sequencing of frames, the transmission of ACKs and NACKs, and maintaining frame ordering and
- Link Control - controlling the state of the link and handling requests for *connect* and *disconnect*.

Each sublayer can be viewed as having its own protocol for communicating with peer sublayers in other entities. However, the boundary between sublayers is not as firm as between layers. For example, the sublayers can act together via shared variables to accomplish their task. This will be further explained in the descriptions of the protocol sublayers.

Each of the sublayer protocols will be described in text and some will be described by *modified transition diagrams* [Boc80]. These diagrams are an extension of the *state transition diagram* and allow transitions to manipulate variables via actions described in fragments of program code. Transitions are labeled and typically identified further as output transitions (underlined) or as non-output transitions (not underlined). A table links a transition label with the program code fragment. Associated with each transition (in the table) is an enabling predicate; the transition may be taken only if the predicate is true. If multiple transition predicates are true, the transition taken is randomly chosen; however, a rule for

arbitrating the decision can be specified. In addition, diagrams may be hierarchically dependent on other (higher level) diagrams. For example, the Data Transfer protocol is dependent on the state of the Link Control protocol, since Data Transfer only applies in the "connected" and "waiting_DISC" states.

To aid in understanding the protocol descriptions, an overview of the protocol layers and the sublayers within the Link layer appears on the next page. This is followed by a figure showing frame layout, "control" character assignments and types of frames.

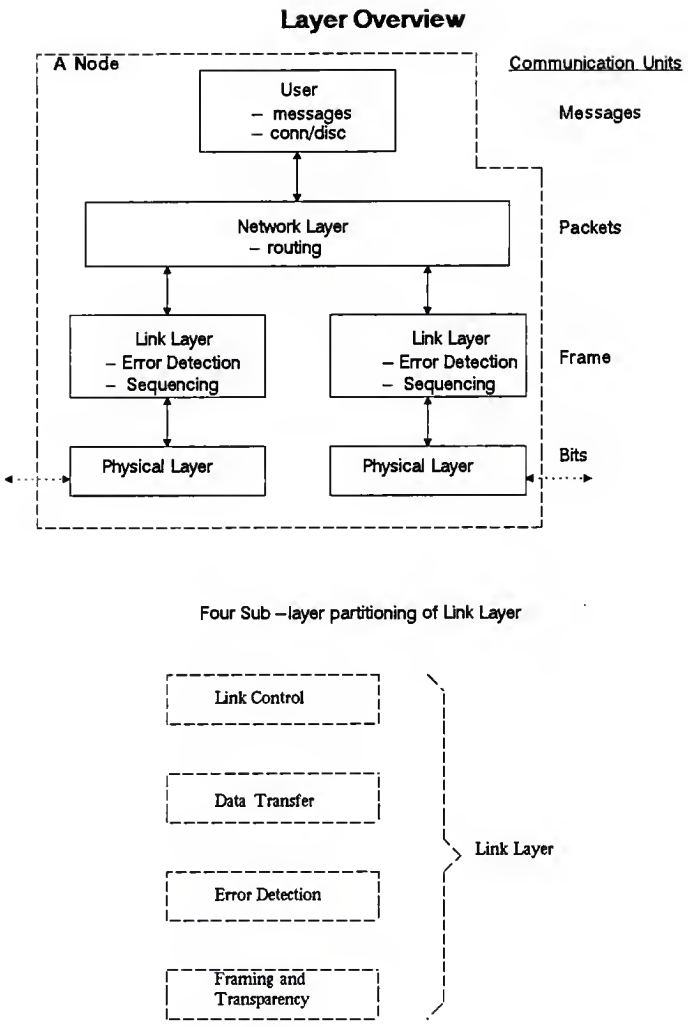
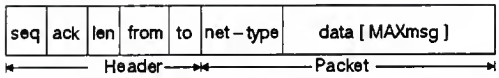


Figure 2. Design Protocol – Overview and Sublayers

Frame Layout



seq = Information Frame (values '0' to '0' + MaxSeq)
or Control Frame (values '0' + MaxSeq + 1 to 255 decimal) [See Below]

ack = Piggy – backed Acknowledgement (values '0' to '0' + MaxSeq)

len = length of Frame (in character units) [3 or more for Control Frames]
[6 or more for Information Frames]

from = ID of frame originator

to = ID of intended destination (0 implies "broadcast frame")

net_type = Network Level Information (type of data in Packet, etc.)

data = 0 to MAXmsg characters of Packet data

Frame in Transmission ← transmission direction

SOH	Frame with ESCaped characters	ESCaped Checksum	EOB

SOH = 0253, EOB = 0252, ESC = 0251 (octal)
Checksum = CCITT standard CRC calculation over non – ESCaped Frame

Frame Types [Control Frame character IDs in ()]

* INFO	Information Frame (contains Packet as above)	Data Transfer (only valid if Link is up)
ACK (A)	Explicit Acknowledgement Frame [sent only if no reverse INFO traffic]	
NACK (N)	Negative Acknowledgement [sent on first bad frame in a sequence of bad frames]	
HEAR (H)	Heartbeat [sent occasionally if no other traffic]	
* START (S)	Start Connection ["ack" unused]	Link Control Frames
STACK (T)	Connection Acknowledgement ["from" contains next valid INFO "seq" value]	
* DISC (D)	Request Disconnetc	
AMDISC (M)	Disconnect Complete ["ack" unused]	

"*" indicates Frame Types that are timed and retransmitted if unacknowledged.

Figure 3. Frame types and layout

3.5 Framing and Transparency Sublayer

The Framing sublayer operates almost identically to Comer's Data Link level processes. The only major changes are the addition of a checksum just before the transmission of EOB and the elimination of the RESTART "control" character. The modified transition diagrams for both the input and output portions of the Framing and Transparency protocol appear in the next two figures.

Note that a RESTART "control" character is not transmitted in the extended protocol. Errors that would have resulted in RESTART will now generate a NACK. However, a related mechanism is implemented using a global variable *G_restart*. The Data Transfer sublayer sets *G_restart* only when beginning to retransmit a series of frames. The Framing sublayer will stop transmission of the current frame (if any) if *G_restart* is set. Framing will also reset *G_restart*. This allows any transmission to be stopped immediately, rather than waiting for it to complete, so that the retransmission can begin sooner.

As an example of this protocol:

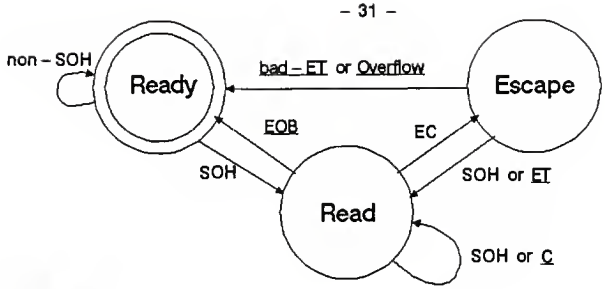
```
            <------ transmission direction

     Header,A,B,ESC,F,SOH                    :original Frame characters

SOH,Header,A,B,ESC,[ESC],F,ESC,[SOH],cksum,EOB      :as transmitted

     Header,A,B,ESC,F,SOH,cksum              :passed to Error Detection
```

where [bracketed] characters represent the original characters with bit 7 set to zero.



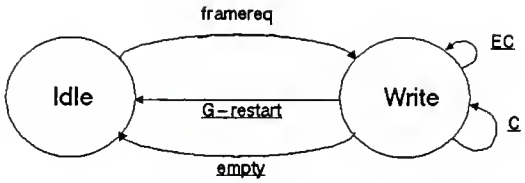
Global
G_restart – set by foutput()
Initial State
Ready
Variables and Functions

count : int	number of characters stored in "buf"
buf[MAXfr] : char array	buffer to hold result frame
ch : char	holds last character read
send(X)	send frame or error status to Error Detection sub – layer
readCH()	reads next character from Physical Layer
unesc(ch)	returns character to substitued for "ch" when following ESC on input (returns – 1 on error)

Note: Prior to testing for the Transition, an implied "ch = readCH()" is executed.
If multiple Enabling Predicates are true, the first listed Transition is used.
The notation "[oldState > newState]" indicates State Transition from diagram above.

Transition	Enabling Pred.	Action	Meaning
SOH ["any" > Read]	ch = = SOH	count = 0	Start processing chars.
non – SOH [Ready]	ch != SOH	–	Wait for it!
EOB [Read > Ready]	ch = = EOB	send(buf)	Send frame up
EC [Read > Escape]	ch = = ESC	–	Saw ESC
C [Read]	!{SOH,EOB,ESC} &&count < = bufsize	buf[count + +] = ch	Store ch
Overflow ["any" > Ready]	count > bufsize	send("error2")	Indicate frame TOO LARGE
ET [Escape > Read]	unesc(ch) > = 0	buf[count + +] = unesc(ch)	Store un – escaped character
bad – ET [Escape > Ready]	unesc(ch) < 0	send("error1")	Indicate BAD char. was found

Figure 4. Framing and Transparency (Input)



Global

G_restart – set by foutput

Initial State

Idle, fr = framereq(), G_restart = false

Variables and Functions

fr: frame pointer	address of frame to transmit
frcount: integer	number of frame characters remaining to transmit
cksum[2]: char	holds checksum in character form
ch: char	holds character from the frame
next()	returns next character from frame
escape(x)	true if x should be escaped
escapec(c)	replacement character for c when escaped
framereq()	returns address of frame to send
crc(fr,count)	returns CCITT CRC checksum

Transitions	Enabling Pred.	Action	Meaning
framereq [idle > Write]	framereq() completes, returning char.	frcount = fr.length cksum = crc(fr,frcount) write SOH ch = next()	Start sending frame
C [Write]	frcount > 0 && !G_restart && !escape(ch)	write ch ch = next() frcount – –	Transmit character
EC [Write]	frcount > 0 && !G_restart && escape(ch)	write ESC ch = escapec(ch) write ch ch = next() frcount – –	Transmit character requiring ESC processing
G_restart [Write > idle]	G_restart true	release fr	Stop early
empty [Write > idle]	frcount < = 0	if(escape(ch = cksum[1])) write ESC ch = escapec(ch) write ch [.. repeat for cksum[2]] write EOB release EOB	Finish sending frame

Figure 5. Framing and Transparency (Output)

3.6 Error Detection Sublayer

The error detection sublayer is responsible for handling errors reported by the Framing and Transparency (input) sublayer and for detecting any other errors in the frame. This protocol does nothing to output, since the Frame and Transparency sublayer computes and sends the checksum. In effect, Error Detection provides a function called *send* that is called by the Framing and Transparency sublayer. A function called *sendon* is then used by Error Detection to pass frame and error indications up to the Data Transfer sublayer. The pseudo-code for the *send* function is shown in the figure below.

```
send(fr,count,status)          fr=frame address
                               count=no. non-ESC characters read
                                     between SOH and EOB.
                               status=0 unless error
                                     detected by Framing sublayer.
{
    if (status != 0) {
          sendon(status);      pass error to Data Transfer
          return;
    }
    count = count - 2;              remove checksum from frame
    if ( count < MINfr ) {
          sendon(4);           frame too short
          return;
    }
    if ( count != fr.len ) {
          sendon(5);           count incorrect
          return;
    }
    if ( crc(fr,fr.len) != checksum at end of frame) {
          sendon(6);           bad checksum
          return;
    }
    sendon(fr);                send good frame to Data Transfer
}
```

Figure 6. Error Detection Diagram

3.7 Data Transfer Sublayer

This sublayer implements the sequencing checks that ensure correct (in-order) delivery of packets to the Network layer. It implements a seven frame sending window and a single frame receive window. The "Go−Back−N" retransmission strategy is used. Operation of the Data Transfer sublayer occurs only when the LinkState (in Link Control) is "connected" or "waiting_DISC."

Four variables control the action of the protocol:

- ExpectedF - this is the sequence number of the next frame that can be acknowledged and sent up to a higher level,
- ExpectedA - this is the sequence number of the oldest unacknowledged frame (if any) waiting for acknowledgement,
- NextF - the sequence number to be associated with the next frame sent over the link and
- NoNack - a boolean that allows a NACK response if true.

In general, ExpectedF is incremented when a frame is sent up to a higher level. ExpectedA is incremented for each unacknowledged frame that is acknowledged by an incoming ACK. NextF is incremented each time a frame is added to the unacknowledged send window (up to 7 outstanding). NextF is set to ExpectedA to begin a "Go−Back−N" retransmission, and the *G_restart* global variable is set. NoNack is initially *true*, becomes *false* if a NACK is sent and then becomes *true* when the ExpectedF frame is finally accepted. (Only one NACK is sent on an error for a particular value of ExpectedF, to avoid swamping the link with long series of NACKs if long series of out-of-sequence frames are being received.)

This strategy of avoiding redundant NACKs has an unfortunate failure mode[13]. While

unlikely, the failure "stalls" the protocol until some reverse traffic (to the transmitter) tells the transmitter the last frame received in sequence (via the piggyback ACK). To "fix" the failure without reverse traffic, the HEARtbeat frame (essentially internal traffic) is transmitted if no reverse traffic appears in a defined interval.

In order to effectively use the piggyback ACK, the explicit ACK frame is delayed for up to three incoming frames (or a short traffic-less interval). Any outgoing traffic during the delay will cancel the explicit ACK.

13. The scenario leading to this state is described in the Data Transfer Failure Scenarios figure later in this chapter. The problem occurs only when there is no traffic in either direction other than the frame being retransmitted. Two failures are also required to force this condition, the loss of the ACK sent when the frame was originally correctly received and the loss of the NACK returned the first time the frame was re-received.

```
        No State Diagram -- just reacts to incoming frames
                            and requests to send new Packets.
Globals
 LinkState:{DISC,waiting_DISC,CONN,waiting_CONN}
                          from Link Control
 ExpectedF:Seq  next incoming frame expected
 ExpectedA:Seq  next acknowledgement expected
 NextF:Seq      next seq. number for outgoing frame
 buffered       total frames buffered in Data Transfer
    (Variables of type Seq use modulo N arithmetic)
    ( N = a small integer, typically a power of 2)

Variables and Functions
 windowed       frames currently in window (max. Nwindowed)
 DoNack:bool    true initially and when good ACK arrives.
                false if ExpectedF already NACKed once
 FR             type of Frame Received {INFO,ACK,NACK,HEAR}
 send(type,num) adds "type" frame to Transmit Queue with
                ack = (ExpectedF-1).  If INFO, num =
                frame number in window.  Starts frame timer
                and ACK timer.  Sets "lastACK" to (ExpectedF-1).
 between(x,y,z) true if x<=y<z, modulo seq. numbers.
 sendup()       sends packet in current frame to Network Layer.

   The following only operate when LinkState={CONN,waiting_DISC}

Frame or event  Action when frame received
--------------  --------------------------

  INFO,seq,ack: if (seq == ExpectedF) {
                    sendup();   /* Frame to higher layer */
                    ExpectedF ++;
                    DoNack = true;
                }
                else if (seq != ExpectedF && DoNack) {
                    send(NACK,0);   /* Send NACK if 1st time */
                    DoNack = false;
                }
   /* The following segment is referred to as HANDLE ACK below */
                while between(ExpectedA,ack,NextF) {
                    remove frame from window
                    buffered --;
                    windowed --;
                    ExpectedA ++;
                    stop timer for frame, start ACK timer
                    add new frame to window from Request Queue,
                    if Request Queue not empty,
                        move from Request Queue to window
                        send(INFO,NextF)
                        NextF++;
                }
   /* End segment called HANDLE ACK */

    ACK:        [ HANDLE ACK as above ]
```

Figure 7. Data Transfer Diagram (Part I)

```
   NACK:       while between(ExpectedA,ack,NextF) {
                   remove frame from window
                   buffered --;
                   windowed --;
                   ExpectedA ++;
                   stop timer for frame
                   if Request Queue not empty,
                       move from Request Queue to window
                       NextF++;  /* DON'T send() yet! */
               }
                   /* Start retransmission here */
               for ( i=ExpectedA; i < NextF; i++ ) {
                   send(INFO,i);
               }

   HEAR,ack:   [ HANDLE ACK as on previous Figure ]
                  /* Currently acts just like ACK */

FrameTimeout:      /* Start retransmission here */
               for ( i=ExpectedA; i < NextF; i++ ) {
                   send(INFO,i);
               }

 ACKtimeout:   /* Slow outgoing traffic, ACK if needed */
               if (lastACK != ExpectedF-1)
                       send(ACK,0);

 HEARtimeout:  /* No outgoing traffic, send HEAR */
               send(HEAR,0)

  fwdmsg:        /* Transmit Packet for higher layer */
         (Allowed only if buffered+priority < MAXbuffered)
               if (windowed < Nwindowed) {
                   add frame to window and send(INFO,Nextf)
                   NextF++;
                   windowed ++;
               }
               else {
                   add frame to Request Queue
               }
               buffered++;

   error:      /* Bad Frame received */
               if (DoNack) send(NACK,0);
               DoNack = false;

 HEARfail:     /* Failure to receive HEARtbeat */
               Print failure message
```

Figure 8. Data Transfer Diagram (Part II)

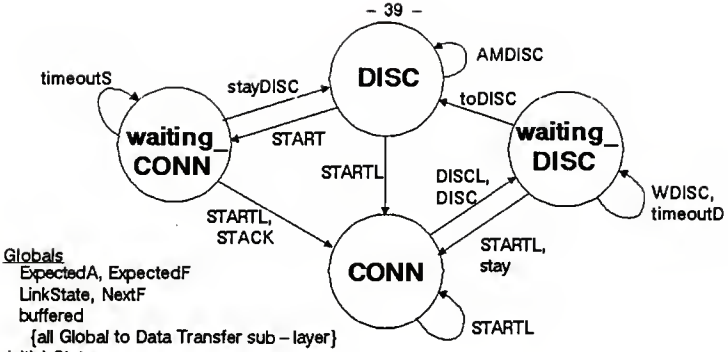
3.8 Link Control Sublayer

Link Control allows the user or other end of the link to request that the link be placed into the "connected" or "disconnected" state. On connect requests, the connection is attempted MAXattempts times before giving up and reporting the problem. The connection will synchronize properly the sequence numbers expected at each end of the link. A connect request from the link to an already connected link is assumed to be from a node that "crashed." The protocol will resequence the sequence numbers such that any unacknowledged frames in the "connected" node will be accepted by the requesting node. The requesting node is required to begin its sequencing at 0, but then a "crashed" node probably lost any unacknowledged frames anyway. User requests to send or receive packets over a "disconnected" link will be refused.

Disconnect requests can originate either from the user or over the link. The timeout interval is fairly long, because the AMDISC acknowledgement is not sent until all unacknowledged frames have been acknowledged. Disconnect requests have a sequencing problem also. It is possible for an unacknowledged frame to be retransmitted many times without an ACK or NACK being returned[14]. If disconnect had to wait until this "lost" frame was acknowledged, a user would be waiting a long time for the request to complete on what should be an idle link! To prevent the long waiting time, an *ack* field appears in the DISC frame to explicitly acknowledge such frames. (The HEARtbeat frame would eventually cure this problem, since it also explicitly acknowledges the last frame received in sequence.)

---

14. This is described in the Data Transfer section as a justification for the HEARtbeat frame.



Globals
ExpectedA, ExpectedF
LinkState, NextF
buffered
{all Global to Data Transfer sub – layer}

Initial State
DISC
NextF, ExpectedF, ExpectedA, buffered = 0

Variables
attempts : integer — counts START or DISC frames sent
FR : frametype — type of Frame received
receivedDISC : {true,false,waiting}

Note: Enabling Predicates are tested in the order listed. Testing occurs only after an event has occurred. Events are "new Frame received" (causing FR to be set), "user transactions", and "timeouts" of various types. Portions of Predicates referring to the FR variable apply only if the event causing evaluation was "new Frame received."

Transition	Enabling Pred.	Action	Meaning
START [DISC > waiting_CONN]	user transaction fcontrol(CONN)	LinkState = waiting_CONN attempts = 0 start STARTtimer ExpectedA, NextF = 0 ExpectedF = 0 < – altered by later STACK ? send START	CONN from user, attempt CONN.
STARTL ["any" > CONN]	FR = = START	LinkState = CONN ExpectedF = 0 stop STARTtime & DISCtimer send STACK(ExpectedA)	CONN from Link, force CONN, Data Transfer will retransmit window (if not empty)
STACK [waiting_CONN > CONN]	FR = = STACK(n)	LinkState = CONN ExpectedF = n Stop START timer	CONNected! "n" is the frame seq. # other end will start using.
AMDISC [DISC]	FR I = START & FR I = AMDISC	send AMDISC	Tell other end we're DISConnected
haveDISC [waiting_DISC]	(FR = = AMDISC or FR = = DISC) & buffered = = 0	receivedDISC = true	Finally got it!! (toDISC may be true now!)

Figure 9. Link Control Diagram (Part I)

Transition	Enabling Pred.	Action	Meaning
timeoutS [waiting_CONN]	STARTtimer timeout & attempts < MAXattempts	attempts + + send START start STARTtimer	Continue CONN attempt.
stayDISC [waiting_CONN > DISC]	FR = = DISC or (STARTtimer timeout & attempts > = MAXattempts)	LinkState = DISC print "Can't CONNect!" Send DISC	Can't CONNect Link!!
DISC [CONN > waiting_DISC]	user transaction fcontrol(DISC)	LinkState = waiting_DISC attempts = 0 receivedDISC = false start DISCtimer	User want's DISC. (WDISC may be true immediately!)
DISCL [CONN > waiting_DISC]	FR = = DISC	LinkState = waiting_DISC attempts = 0 receivedDISC = true start DISCtimer	Link want's DISC. (toDISC may be true immediately!)
timeoutD [waiting_DISC]	DISCtimer timeout & attempts < MAXDattempts	attempts + + send DISC start DISCtimer	Continue DISC attempt.
stay [waiting_DISC > CONN]	DISCtimer timeout & attempts > = MAXDattempts	LinkState = CONN print "Can't DISConnect!" if(buffered = = 0) ExpectedA = NextF = 0 send START	Can't DISconnect link! (Send START if possible to force other end to CONN)

Note: The following Transitions are tested after any of the preceeding Actions and when (buffered = = 0) is true [due to Data Transfer sub - layer action]. In effect, "buffered = = 0" is an "event" that causes the Predicates to be tested.

Transition	Enabling Pred.	Action	Meaning
WDISC [waiting_DISC]	buffered = = 0 & receivedDISC false	send DISC receivedDISC = waiting	Ready to tell other end to DISC.
toDISC [waiting_DISC > DISC]	buffered = = 0 & receivedDISC true	LinkState = DISC send AMDISC stop DISCtimer ExpectedF = NextF = 0 ExpectedA = 0	Finally DISConnected!!

Figure 10. Link Control Diagram (Part II)

3.9 Operation Scenarios

The remainder of this chapter demonstrates the performance of the various sublayers just described. No attempt has been made to show all possible sequences within each sublayer. However, all non-trivial transitions within each of the Link Control and Data Transfer sublayers are used by at least one scenario. The scenarios use a time sequence diagram to show the transmission and reception of frames and their relationship to the peer sublayer at the other end of the link.

The diagram is read from the top down. The left side of page represents one end of the link, the right represents the other. Across the page, the first column represents the state of the Data Transfer or Link Control variables of interest in that scenario. Next is a field describing the frame being sent or received. The center (usually white) column represents the physical link, with comments occasionally indicating something other than normal transmission. The remaining two columns represent the frame being sent or received and the state variables of interest, respectively, for the other end of the link. Events occurring at the same time or in random order are represented by having them appear on the same line of the diagram.

A shorthand method of describing each frame is used. The frame type is identified by its acronym (see "Frame types and layout" above), followed by the sequence number and acknowledgement number, where applicable. Since the sequence number applies only to INFO frames, any number following other frame types is the acknowledgement number.

A "+" before the frame type indicates reception of that frame, a "−" indicates the frame is being transmitted. The value (before reception or transmission) of the Data Transfer

variables [ExpectedF, ExpectedA and NextF] are represented in some instances as "[x,y,z]" on the appropriate side of the page, with "−" indicating a "don't care" value of the associated variable. The center of the page represents the physical link, normally blank. An **X** indicates the associated frame was damaged in transit while an **L** indicates the frame was lost.

Another shorthand notation indicates that the user's message was removed from the "unacknowledged frame" buffer. This is represented as "−msgN," typically next to the reception of an ACK, where N represents the frame sequence number. The notation "−hostN" will be used to indicate that a received frame with sequence number N has been passed up to the Network routing process (in order and without duplication).

Light traffic – single INFO, ACK, then single INFO, return INFO with piggyback ACK, ACK

Bracketed Variables are [ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}

	Node A	Physical Link	Node B	
EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >	EF,EA,NF
[0,0,0]		{Initial State}		[0,0,0]
[0,0,1]	– INFO0,7			
			+ INFO0,7	[1,0,0] – host0
	{B receives Frame in sequence,sends to Network Layer ("host")}			
			< ACKtimeout >	
			– ACK,0	
[0,1,1] – msg0	+ ACK,0			
	{Link quiet, time passes...}			
[0,1,2]	– INFO1,7			
			+ INFO1,7	[2,0,0] – host1
	{Reverse traffic ACKs the INFO1,7 Frame}			
			– INFO0,1	[2,0,1]
[1,2,2] – msg1 – host0	+ INFO0,1			
	< ACKtimeout >			
	– ACK,0			
			+ ACK,0	[2,0,1] – msg0
	{Link quiet, time passes...}			

Figure 11. Normal Data Transfer – Light Traffic

Maximum 1 – way traffic – 7 INFOs, then window full, ACK, more INFOs

Bracketed Variables are [ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}

	Node A	Physical Link	Node B	
EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >	EF,EA,NF
[0,0,0]		{Initial State}		[0,0,0]
[0,0,1]	– INFO0,7			
			+ INFO0,7	[1,0,0] – host0
	{B receives Frame in sequence,sends to Network Layer ("host")}			
[0,0,2]	– INFO1,7			
[0,0,3]	– INFO2,7			
[0,0,4]	– INFO3,7	{A sends a burst of INFOs}		
[0,0,5]	– INFO4,7			
[0,0,6]	– INFO5,7			
[0,0,7]	– INFO6,7	{Window full, wait for incoming Frames from B}		
			+ INFO1,7	[2,0,0] – host1
			+ INFO2,7	[3,0,0] – host2
			+ INFO3,7	[4,0,0] – host3
			+ INFO4,7	[5,0,0] – host4
			+ INFO5,7	[6,0,0] – host5
			+ INFO6,7	[7,0,0] – host6
		{A's window is still full...}		
			< ACKtimeout >	
			– ACK,6 {ExpectedF – 1}	
[0,7,7]	+ ACK,6			
	– msg0, – msg1..., – msg6 {Remove messages from window}			
	{If there are Frames in the Request Queue, up to 7 more of them will be sent by ACK HANDLER Action}			

Figure 12. Normal Data Transfer – Max. one – way Traffic

Maximum 2 – way traffic – INFOs in each direction ACKing prior INFOs

Bracketed Variables are [ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}

Node A		Physical Link	Node B	
EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >	EF,EA,NF
[0,0,0]		{Initial State}		[0,0,0]
[0,0,1]	– INFO0,7			
			– INFO0,7	[0,0,1]
[1,0,1] – host0	+ INFO0,7		+ INFO0,7	[1,0,1] – host0
		{Neither INFO has yet been ACKed}		
[1,0,2]	– INFO1,0	{Each sends piggyback	– INFO1,0	[1,0,2]
[1,0,3]	– INFO2,0	ACKs for INFO0}	– INFO2,0	[1,0,3]
[2,1,3] – host1	+ INFO1,0			
[3,1,3] – host2	+ INFO2,0		+ INFO1,0	[2,1,3] – host1
			+ INFO2,0	[3,1,3] – host2

{Now Frames 1 and 2 are unacknowledged}
{Further traffic or ACKtimeouts will acknowledge them}

Figure 13. Normal Data Transfer – Max. two – way Traffic

Light traffic – single INFO, lost, Frame timeout

Bracketed Variables are [ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}

	Node A	Physical Link	Node B	
EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >	EF,EA,NF
[0,0,0]		{Initial State}		[0,0,0]
[0,0,1]	– INFO0,7			
		Lost Frame		
	< Frame timeout >			
	– INFO0,7			
			+ INFO0,7	[1,0,0] – host0
			< ACKtimeout >	
			– ACK,0	
[0,1,1] – msg0	+ ACK,0	{Finally got acknowledgement}		

Figure 14. Data Transfer Failure – lost INFO/ACK

Frame timeout occurs before ACK gets back to Frame's sender

Bracketed Variables are [ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}

	Node A	Physical Link	Node B	
EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >	EF,EA,NF
[0,0,0]		{Initial State}		[0,0,0]
[0,0,1]	– INFO0,7			
			+ INFO0,7	[1,0,0] – host0
			< ACKtimeout >	
	< Frame timeout >		– ACK,0	
	– INFO0,7			
	{Frame timeout interval is so short that the ACK from B hasn't come back yet...}			
[0,1,1] – msg0	+ ACK,0 {ACK finally arrives}			
	{So does the re – transmitted INFO}		– INFO0,7 {Out – of – sequence!!}	
			– NACK,0 {DoNack now false}	
	+ NACK,0 {NACK does NOT cause retransmission}			

Figure 15. Data Transfer Failure – Short Timeout

Heavy 1 – way traffic with 1 lost INFO

Bracketed Variables are [ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}

Node A		Physical Link	Node B	
EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >	EF,EA,NF
[0,0,0]		{Initial State}		[0,0,0]
[0,0,1]	– INFO0,7			
[0,0,2]	– INFO1,7			
[0,0,3]	– INFO2,7		+ INFO0,7	[1,0,0] – host0
[0,0,4]	– INFO3,7	Lost INFO3!		
[0,0,5]	– INFO4,7		+ INFO1,7	[2,0,0] – host1
			+ INFO2,7	[3,0,0] – host2
			+ INFO4,7 {Out – of – Sequence!!}	
			– NACK,2 {DoNack now false}	
[0,3,5] – msg0	+ NACK,2	{NACK piggyback ACK allows INFO0/1/2 to be removed from}		
– msg1	– INFO3,7	{window, and causes INFO3 and up to be re – transmitted.}		
– msg2	– INFO4,7			
			– INFO3,7	[4,0,0] – host3
			– INFO4,7	[5,0,0] – host4
			< ACKtimeout >	
			– ACK,4	
[0,5,5] – msg3	+ ACK,4			
– msg4		{Finally, all INFOs ACKnowledged}		
		{Link quiet once again}		

Figure 16. Data Transfer Failure – lost INFO in Heavy Traffic

Heavy Traffic, several lost INFOs

Bracketed Variables are [ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}

Node A		Physical Link	Node B	
EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >	EF,EA,NF
[0,0,0]		{Initial State}		[0,0,0]
[0,0,1]	– INFO0,7			
[0,0,2]	– INFO1,7			
[0,0,3]	– INFO2,7		+ INFO0,7	[1,0,0] – host0
[0,0,4]	– INFO3,7	Lost INFO3I		
[0,0,5]	– INFO4,7	Lost INFO4I	+ INFO1,7	[2,0,0] – host1
[0,0,6]	– INFO5,7		+ INFO2,7	[3,0,0] – host2
			+ INFO5,7 {Out – of – SequenceII}	
			– NACK,2 {DoNack now false}	
[0,3,6] – msg0	+ NACK,2	{NACK piggyback ACK allows INFO0/1/2 to be removed from}		
– msg1	– INFO3,7	{window, and causes INFO3 and up to be re – transmitted.}		
– msg2	– INFO4,7			
	– INFO5,7			
		Lost INFO4I		
			+ INFO3,7	[4,0,0] – host3
			{INFO in sequence, so DoNack is true.}	
			+ INFO5,7 {Out – of – SequenceII}	
			– NACK,3 {DoNack now false}	
[0,4,6] – msg3	+ NACK,3			
	– INFO4,7			
	– INFO5,7			
			+ INFO4,7	[5,0,0] – host4
			+ INFO5,7	[6,0,0] – host5
			< ACKtimeout >	
			– ACK,5	
[0,6,6] – msg4	+ ACK,5			
– msg5		{Finally, all INFOs ACKnowledged}		
		{Link quiet once again}		

Figure 17. Data Transfer Failure – Many lost INFOs

Light 1 –way traffic – single INFO, lost ACK and NACK

Bracketed Variables are [ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}

	Node A	Physical Link	Node B	
EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >	EF,EA,NF
[0,0,0]		{Initial State}		[0,0,0]
[0,0,1]	– INFO0,7			
			+ INFO0,7	[1,0,0] – host0
			< ACKtimeout >	
			– ACK,0	
		Lost ACK		
	< Frame timeout >			
	– INFO0,7			
			+ INFO0,7 {Out – of – Sequence!!}	
			– NACK,0 {DoNack now false}	
		Lost NACK		
	< Frame timeout >			
	– INFO0,7			
			+ INFO0,7 {Still bad seq. number.}	
			{DoNack false, don't send NACK}	
	< Frame timeout >			
	– INFO0,7			
			+ INFO0,7 {Still bad seq. number.}	
			{DoNack false, don't send NACK}	
	{Protocol is "stalled" due to loss of a NACK after ACK/piggyback ACK}			
	{INFO has actually arrived, and a new INFO from either direction will}			
	{"unstall" the protocol, so the only real problem here is that the Link is}			
	{incurring a lot of wasted re – transmission. A HEARtbeat from B to A}			
	{eventually fixes things, but actually almost any kind of Frame would do.}			
			< HEARtimeout >	
			– HEAR,0	
[0,1,1] – msg0	+ HEAR,0			
	{Link quiet once again}			

Figure 18. Data Transfer Failure – Stalled without Traffic

Normal CONNect request while in DISC state

Variables are [LinkState(LS),ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}. "w_DISC" means "waiting_DISC; ditto for "w_CONN".

Node A		Physical Link		Node B	
LS,EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >	LS,EF,EA,NF
[DISC,0,0,0]		{Initial State}		[DISC,0,0,0]
[w_CONN,0,0,0]	< fcontrol(CONN) > – START			
			+ START – STACK,0	[CONN,0,0,0]
[CONN,0,0,0]	+ STACK,0			
		{The Link is now "up"!!}		

Figure 19. Normal Link Control – Connect

Normal DISConnect request while in CONN state

Variables are [LinkState(LS),ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}. "w_DISC" means "waiting_DISC; ditto for "w_CONN".

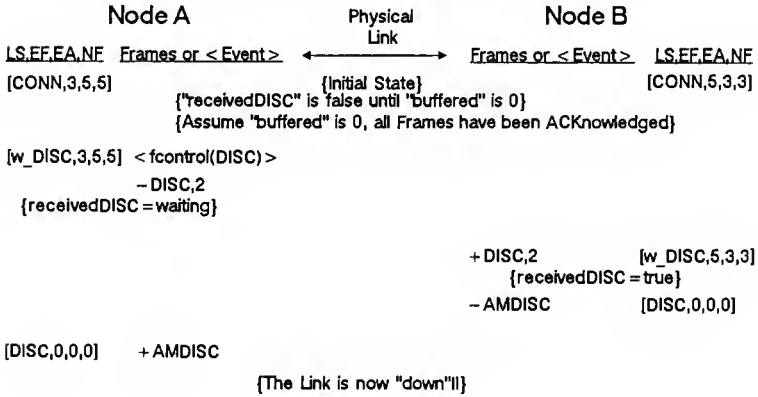


Figure 20. Normal Link Control – Disconnect

Normal DISConnect request while in CONN state – unacknowledged Frames

Variables are [LinkState(LS),ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}. "w_DISC" means "waiting_DISC; ditto for "w_CONN".

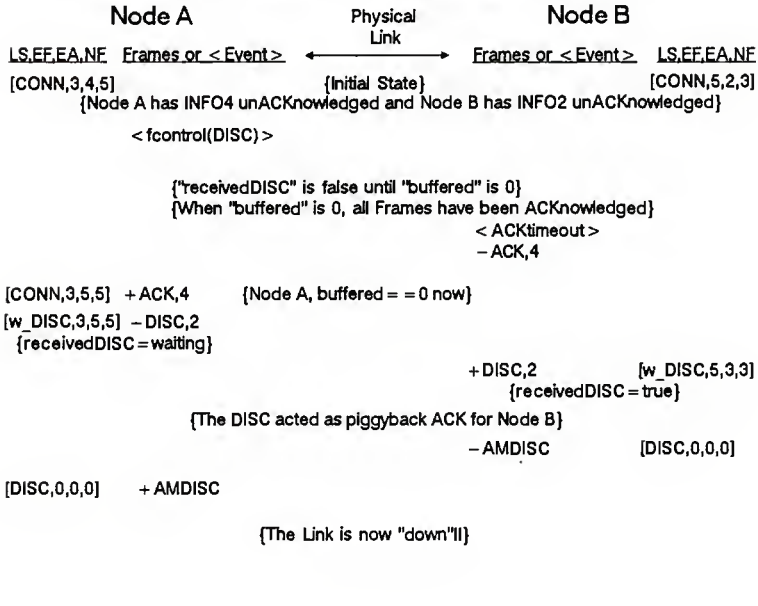


Figure 21. Normal Link Control – Disconnect with Traffic

Incoming CONN while in "CONN" state – Could be "crashed" node coming up

Variables are [LinkState(LS),ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}. "w_DISC" means "waiting_DISC; ditto for "w_CONN".

Node A		Physical Link		Node B	
LS,EF,EA,NF	Frames or < Event >	⟷	Frames or < Event >		LS,EF,EA,NF
[CONN,4,6,7]		{Initial State}			[DISC,0,0,0]
		{Node B just coming up after "crash", attempts CONN}			
			< fcontrol(CONN) >		
			– START		[w_CONN,0,0,0]
[CONN,0,6,7]	+ START {INFO6 has not been acknowledged}				
	– STACK,6				
	– INFO6,7				
			+ STACK,6		[CONN,6,0,0]
			+ INFO6,7		[CONN,7,0,0]
			< ACKtimeout >		
			– ACK,6		
[CONN,0,7,7]	+ ACK,6				
	– msg0				

{INFO Frame 6 survived the crash of Node B}

Figure 22. Link Control Failure – Stray Connect

CONNect request with lost START

Variables are [LinkState(LS),ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}. "w_DISC" means "waiting_DISC; ditto for "w_CONN".

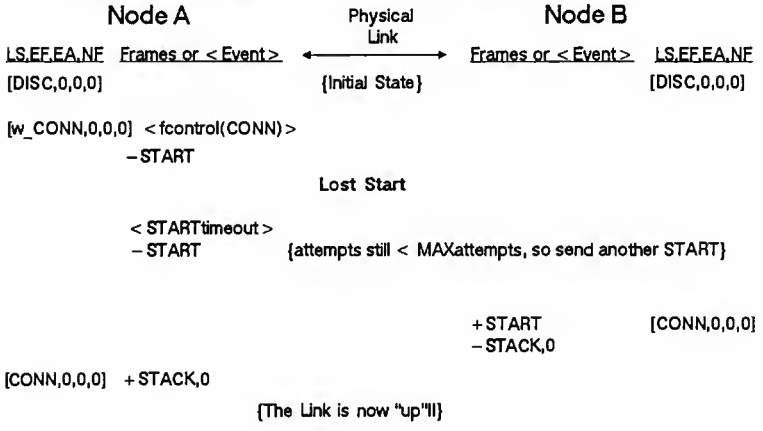
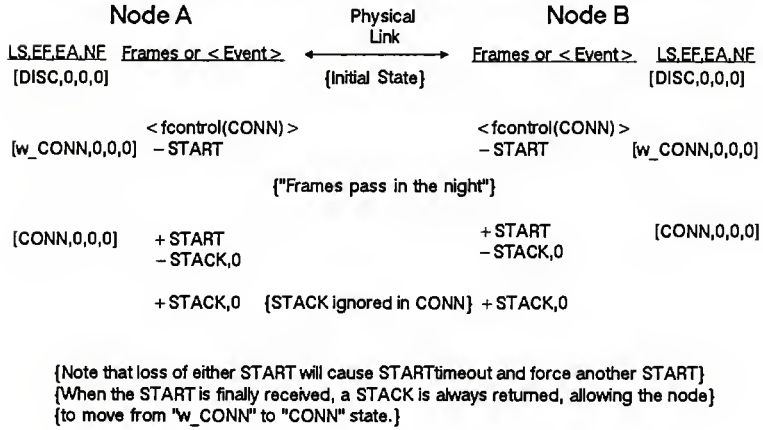


Figure 23. Link Control Failure – Lost START

CONNect from both ends while in DISC state

Variables are [LinkState(LS),ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or < event > is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). < Events other than Frame transmission/reception are shown like this. > and {Comments are shown this way.}. "w_DISC" means "waiting_DISC; ditto for "w_CONN".

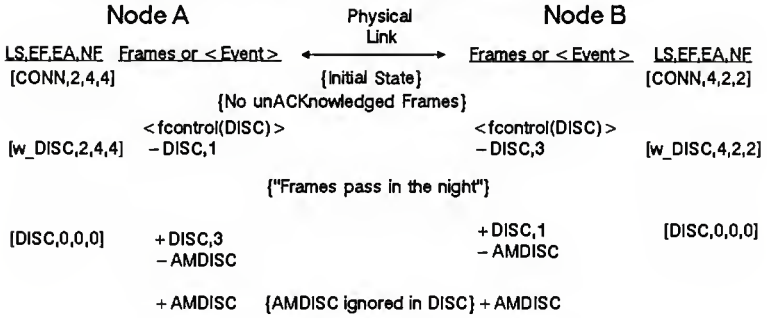


{Note that loss of either START will cause START timeout and force another START}
{When the START is finally received, a STACK is always returned, allowing the node}
{to move from "w_CONN" to "CONN" state.}

Figure 24. Link Control Failure – Double Connect

DISConnect from both ends while in CONN state

Variables are [LinkState(LS),ExpectedF(EF), ExpectedA(EA), and NextF(NF)]. They are only shown when a change in value occurs and reflect the values after the Frame or <event> is processed(i.e., "Action" portion of applicable transition diagram(s) has/have finished.). <Events other than Frame transmission/reception are shown like this.> and {Comments are shown this way.}. "w_DISC" means "waiting_DISC; ditto for "w_CONN".



{Note that loss of either DISC will cause DISCtimeout and force another DISC}
{When the DISC is finally received, an AMDISC is always returned, allowing the node}
{to move from "w_DISC" to "DISC" state.}

Figure 25. Link Control Failure – Double Disconnect

4. IMPLEMENTATION

Implementation of the protocol from the previous chapter is in the Concurrent C [Geh84] language running on a VAX 11/780 under the 4.2 Berkeley Software Distribution version of UNIX™. The choice of the implementation language was made with the knowledge that Concurrent C was not (yet) well-known and that transportation of the protocol to other systems would not be trivial. However, Concurrent C does provide an easily understood method of process interaction. Many of the transitions appearing in the modified state transition diagrams can be mapped into Concurrent C in an obvious manner. Thus the leap from diagram to code is not so difficult, and there is less chance of incorrectly translating a protocol description into code.

4.1 Concurrent C Facilities

A short summary of the facilities provided in Concurrent C will aid in understanding the implementation (and the comments in this chapter). Concurrent C allows processes to interact through a concept called the extended rendezvous; two processes request the rendezvous, synchronize and exchange information and then continue their concurrent operation. The two processes are not equals during a rendezvous. One process requests the rendezvous while the other accepts the request; the request and accept can be executed asynchronously and each will wait on the other. However, the accepting process is allowed to accept requests from multiple processes and may even have multiple types of accepts all open at the same time. The requesting process is allowed only one pending request to one

UNIX is a trademark of AT&T Bell Laboratories.

process at any one time. It is possible to revoke the request, but typically the requester is blocked until the request is accepted[15].

In addition to the Concurrent C language, a facility called the Concurrent C Window Manager [Smi84] is provided. On UNIX, the Window Manager allows controlled interaction with various processes within a Concurrent C program from a single display terminal. Input can be directed to individual processes, even though several may be requesting input simultaneously. Output is directed to virtual windows, and the Window Manager controls which virtual windows appear on the terminal at any given time.

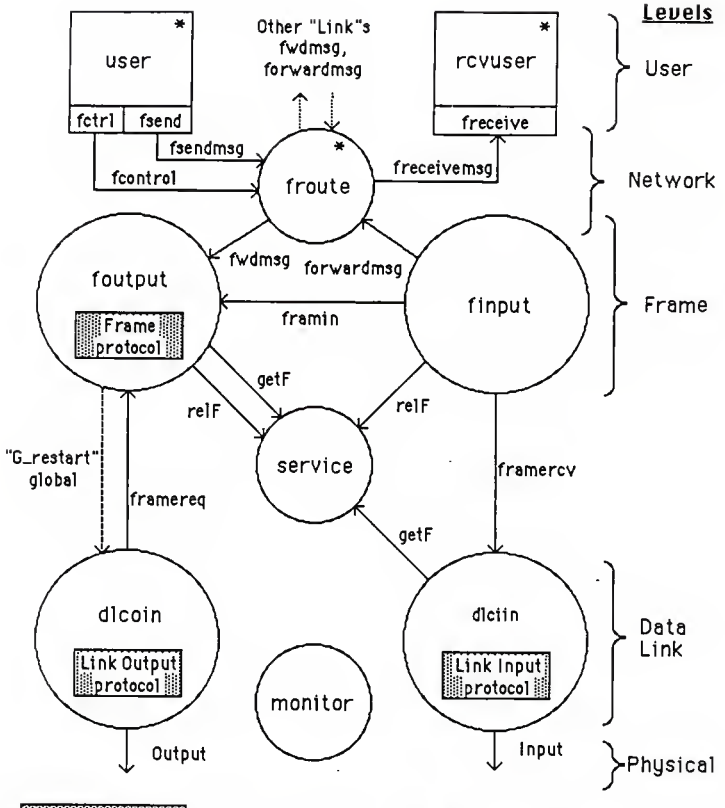
4.2 Implementation Structure

The approach of mapping each protocol sub-layer into Concurrent C processes was not used in this implementation. Such a mapping would require the use of monitors or buffer processes between many of the protocol processes to prevent blockage. Efficiency would suffer from the operation of several small protocols on each frame. Therefore, the sublayers were combined back into the two levels that Comer used, a Frame level and a Data Link level. The processes bear the same names as Comer's Link layer processes, but communicate via Concurrent C transactions rather than global data structures, semaphores and interprocess messages.

The processes also perform somewhat different tasks at the Frame level than the corresponding processes in Comer's implementation. Rather than using separate input and output processes cooperating to implement the Frame level protocol (as in Comer), the

15. There may be an analogy in male/female protocols (social etiquette in *accepting* invitations), but I will leave that investigation to others. (See [Pos55] or, for a more formal treatment, [McC77].)

Frame level protocol is placed entirely within one process, *foutput*. An *finput* process is still used, but only to retrieve incoming frames from the Data Link layer, notify *foutput* and send acknowledged frames up to the Network layer. The result of this shuffling is called the Implementation Model. A diagram of the processes and protocol placement follows.



Frame protocol combined Link Control and Data Transfer protocol.

Link Output protocol Framing and Transparency (output) protocol.

Link Input protocol combined Framing and Transparency (input) and Error Detection protocol.

Labeled arcs are Concurrent C transactions, the accepting process is at the arrowhead end of the arc.

Figure 26. Implementation Model

4.3 Implementation Model

The Implementation Model (just "Model" from now on) provides a fixed set of services between a limited number of processes and assumes that the protocol will fit into the indicated places without resorting to the use of global variables or communication paths other than those shown. Thus, while the Model can certainly support the extended protocol described in the previous chapter, it can also support alternative protocols. Appendix 1 shows the pseudo-code for the Concurrent C processes implementing the Model. Where [... protocol] is shown, the appropriate portion of the protocol is executed.

In general, the Model operates by accepting packets from user processes and the *froute* process, and delivers them to the *dlcoin* process for transmission. The *finput* process waits on *dlciin* to read a packet (or indicate reception of a damaged packet), then passes the frame to *foutput* for analysis. The status returned by *foutput* determines whether the frame is passed up to the *froute* process. The heart of the retransmission and sequencing of frames is controlled by *foutput*. The *service* process is a concession to efficiency. It provides a frame management service that allows frame addresses to be passed between the processes at the Frame and Data Link levels, instead of copying the data.

4.4 Implementation

The protocols from the previous chapter are implemented within the Model using, where appropriate, code that reflects the state transition diagrams. It is thus not difficult to determine that the protocol code matches the diagrams closely. Many of the predicates placed on transitions in the diagrams are handled by *guards* and other facilities provided in Concurrent C.

Appendix 2 contains the code for the implementation, as well as supporting files.

4.5 Test Facilities

To allow control and monitoring of various processes in the Model, a process called *Monitor* was implemented. This process reads test control input from a Virtual Window and acts upon that input. Associated with each link's processes and with the local network layer's processes is a block of *Monitor* variables accessible only by those processes and by the *Monitor* process. The input to *Monitor* is one of the following (where *var* is the name of a *Monitor* variable):

~Mx	- enter *Monitor* variable access mode for the nodeID or linkID specified by "*x*" (required before the following inputs apply),
d var	- display the value of the *Monitor* variable *var*,
d	- display the value of all *Monitor* variables,
s var	- set *var* to 1,
r var	- reset *var* to 0 and
Q	- exit *Monitor* access mode.

Dump, set and reset are rather primitive, but provide the means to control and monitor the other processes. This is accomplished by placing some "instrumentation" code within each of the processes. The instrumentation code allows limited access to the *Monitor* variables[16]. Each variable is used in one of three ways; as a counter, picture or flag.

A counter variable is incremented each time the

```
COUNT(var);
```

16. Concurrent C does not allow access directly to global variables from a process, nor is a process such as *Monitor* allowed access to the local variables of other processes. The only mechanism (besides transactions) for communicating between processes is global variables, but they must be accessed by functions defined outside the processes.

statement appears in the Concurrent C code. The counter variables (and the others) are initially 0.

A picture variable provides a means to tell *Monitor* an (integer) value last computed at some point within a process. For example,

```
PICTURE(var,value);
```

takes a snapshot by placing "value" into the *Monitor* variable "var."

A flag variable controls the execution of alternative code, based on the variable being "set" or "reset." For example, to simulate the effect of a lost ACK when "var" is "set,"

```
if ( ! FLAG(var) ) {
      send ACK;
 }
```

or to block a process from further execution,

```
while( FLAG(var) ) delay 1;
```

The *Monitor* variables are defined at the beginning of each compiled unit in a complex structure and must be initialized there. This requirement is most conveniently met through the use of macros defined in the C header file "monvars.h".

Since *Monitor* can dump and change variables as the processes execute, it can, with intelligent placement of "instrumentation," be used to trace a process's execution and to control dynamically the failure testing of a protocol.

4.6 Protocol Testing

The protocol's processes (collectively called *Proto*) can be tested by executing *Proto* twice (from two separate display terminals). In this instance, two arguments must be provided to *Proto*, one is the nodeID of this instance of *Proto* and the other combines the linkID used by

Proto to identify the (single) link and the name of the I/O port acting as this end of the physical link. The ports must be physically connected together, of course. An example of input to begin a typical test session is:

```
Proto  A 2,/dev/tty3      [Node A reaches Node 2 using device tty3]

Proto  2 A,/dev/ttyi4     [Node 2 reaches Node A using device ttyi4]
```

implying that the tty3 and ttyi4 ports are physically connected together. Many other test configurations are possible; refer to the manual pages for "Proto."

After initialization, the Window Manager will display the execution status of some of the processes on the terminal screen. The contents of the first four virtual windows will also be displayed. Packet transmission is initiated through input to the virtual window associated with a *user* process (or the *Monitor* window). Access to *Monitor* variables is through input to its virtual window as previously described.

4.7 Implementation Limitations

The implementation "works around" several deficiencies in the parts from which it is constructed. UNIX, in general, does not allow controlled inter-process communication between arbitrary user processes. This was one of the reasons for choosing Concurrent C as an implementation language. On the other hand, Concurrent C cannot currently maintain the effect of concurrent processing when requests to the operating system block execution of the underlying UNIX user process. This limitation (concurrency during operating system requests) requires, for example, the use of non-blocking *read* requests from the input device. (This forces a "polling" approach to read requests, something that non-concurrent languages

must typically do.) Protocols that require a multiplicity of readers or other blocking requests to UNIX will have difficulty fitting into the Model *and* achieving concurrent operation. One can hope that improvements in the implementation of Concurrent C would allow "true" concurrency at the UNIX process level, perhaps even maintaining the flexibility that the Window Manager provides.

Also, the extended rendezvous process interface of Concurrent C (and ADA) has difficulty handling the asynchronous event and message passing that is available in a "process control" or message-based environment. By suitable use of several "server" and "buffer" processes, these difficulties may be circumvented, but the Implementation Model cannot claim to handle such needs.

The Model itself, given the above limitations, makes further assumptions about the way that various processes communicate. For example, the user-level process receiving incoming messages is assumed never to communicate anything back down to the Link layer. To allow *freceive* to make flow control requests or to reject frames (they have already been acknowledged) would require some major surgery on the Model.

5. CONCLUSION

This report has presented an extended protocol for the Link layer of a computer communications network. While based on an already-published protocol for ring networks, the extensions have removed the requirement for a ring architecture, improved error detection, increased utilization of the link, and added a degree of control over the link. The result is a protocol that provides a solid base for further extensions or for supporting development of higher protocol layers.

The use of the *monitor* in the implementation, together with the facilities provided by the Window Manager, allows the execution of the protocol to be observed and controlled from a single terminal representing the processes operating at one end of a link. This provides a more satisfying and useful test facility than batch test scripts and reams of printed output tracing protocol execution.

As the applications for computers increase and their need for information grows, it is likely that the need for usable, efficient protocols and implementations will increase. While the protocol and implementation described in this paper are useful and reliable, it cannot be claimed that the ultimate Link protocol has been achieved. The work does, however, provide a foundation for development by others.

5.1 Future Efforts

There are several areas related to this report that are worthy of investigation. First, formal verification of the protocol would provide some solid evidence of the soundness of the protocol. Second, Chapter 2 itemized several possible extensions that were not designed into the protocol. Some of these, in particular the Virtual Circuit capability and the remote

control of nodes, would be major undertakings. Designing and implementing the Network layer routing process would also be an interesting exercise.

In the implementation area, it would be informative to see this Concurrent C implementation compared to an implementation in other languages supporting concurrency. Of course, an implementation in a non-concurrent environment using the protocol descriptions in Chapter 3 would also be possible. One interesting non-concurrent environment would be a micro-computer or work-station, with all communication with other computers using the protocol. Mapping the design into a tty protocol within the UNIX operating system kernel would produce an efficient version of the protocol. An alternative would be placement of the *dlciin* and *dlcoin* processes into the kernel (or a programmable I/O device), with the Frame level remaining as user processes.

It has been said that every communication protocol evolves toward more complexity until it ceases to be used. As a project for those believing in the phrase "small is beautiful," some of the extensions could be removed or revised and the resulting protocol compared to the original.

ACKNOWLEDGEMENT

I would like to thank several individuals for their support in the preparation of this report. The advice and assistance of Dr. Richard McBride made the preparation of this report an educational and even enjoyable experience. My supervisor granted me this time undisturbed by any work-related emergencies. And my wife, Linda, and my family have provided the encouragement and sacrifice needed to allow this effort to take place.

REFERENCES

Amy38 Amyot, (initials unknown) "Note Historique," *Comptes Rendus Acad. des Sciences*, pp. 80-3, July 9, 1838.
(quoted from [Fah74])

Bla83 Black, Uyless D. *Data Communications, Networks and Distributed Processing*, Reston, Virginia: Reston Publishing Co., 1983

Boc79 Bochmann, Gregor v. *Architecture of Distributed Computer Systems*, (Lecture Notes in Computer Science), Berlin, Gemany: Springer-Verlag.

Boc80 Bochmann, Gregor v. "A General Transition Model for Protocols and Communication Services," *IEEE Trans. on Communication*, Vol. COM-28, No. 4, April 1980, pp. 643-50.

Com84 Comer, Douglas. *Operating System Design, the XINU approach*. Prentice-Hall, 1984.

Con80 Conrad, James W. "Character-Oriented Data Link Control Protocols," *IEEE Trans. on Communication*, Vol. COM-28, No. 4, April 1980, pp. 445-54.

Dav83 Davies, Donald Watts. *Computer Networks and their Protocols*, John Wiley & Sons, 1983.

Geh84 Gehani, N. H. and Roome, W. D. *Concurrent C*, Internal Memorandum, AT&T Bell Laboratories, September 1, 1984.

Fah74 Fahie, J. J. *A History of Electric Telegraphy to the Year 1837*. New York: Arno Press, 1974.
(Reprint from original published London: E. & F. N. Spon, 1884)

Gam97 Gamble, Rev. J. *Essay on the Different Modes of Communication by Signals*, London 1797.
(quoted from [Fah74])

Hui81 Huitema, C. and Valet, I. "An Experimental High Speed File Transfer using Satellite Links," *Proceedings - Seventh Data Communications Symposium*, October 27-29, 1981, pp. 254-7.

IEE84 IEEE Computer Society. *ANSI/IEEE Standard for Local Area Networks, IEEE Standard 802.2, Logical Link Control*, December 1984.

Lai82 Lai, Wai Sum. "An Analysis of Piggybacking in Packet Networks," *Computer Networks*, Vol. 6, No. 4, September 1982, pp. 279-90.

McC77 McCaffree, Mary Jane and Innis, Pauline. *Protocol: The Complete Handbook of Diplomatic, Official and Social Usage*, Englewood Cliffs, New Jersey: Prentice-Hall, 1977.

Ner84 Neri, G., Morling, R. C. S., Cain, G. D., Faldella, E., Longhi-Gelati, M., Salmon-Cinotti, T. and Natali, P. "MININET: A Local Area Network for

Real-Time Instrumentation Applications," *Computer Networks*, Vol. 8, No. 2, 1984, pp. 279-90.

Pos55 Post, Emily. *Etiquette, the Blue Book of Social Usage*, New York: Funk & Wagnalls Co., 1955.

Pos81 Postel, J. (ed.), "Internet Protocol - DARPA Internet Program Protocol Specification," *RFC 791*, USC/Information Sciences Institute, September 1981.

Smi84 Smith-Thomas, B. *The Concurrent C Window Manager*, Internal Memorandum, AT&T Bell Laboratories, September 1, 1984.

Tan81 Tanenbaum, A. S. *Computer Networks*. Prentice-Hall, 1981.

Zim80 Zimmermann, H. "OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, Vol. COM-28, pp. 425-432, April 1980.

APPENDIX 1

Original Pseudo-code for the Concurrent C Implementation Model

```
                        /* NOTE:  The first 2 pages appear only for referance */
                        /*        purposes -- they were copied from "proto.h" */
                        /*        early in the design/code phase.             */

/* Universal Constants */
#define MAXpkt 150      /* Max. size of packet with header. */
#define MAXlinks 3      /* Max. no. of Links out of this node */
#define MAXseq   7      /* Highest sequence no. */
#define MAXfr_l (MAXseq+2+4) /* Max. frames per link */

#define ESC     0251
#define EOB     0252
#define SOB     0253
#define RESTART 0257
#define ESCmask 0370
#define ESCbit  0200
#define BROAD   0       /* Destination implying Broadcast */

/* Universal Terms */
#define PUBLIC extern
#define PRIVATE static
#define LOOP for(;;) {
#define ENDLOOP }
#define FNULL (Frametype *)0
  /* Minimum frame address is 50 to allow 0 to 49 as arror codes */
#define FMIN (Frametype *)50

/* Universal Codes */
enum Status { NoFrame
      };
enum UError { U_OK, U_Disc, U_NoLink, U_BadArgs };
enum Errorcode { NoError,
      IGNORE,           /* foutput -> finput => ignore frame */
      ROUTEMSG          /* foutput -> finput => forward to Router */
      };
enum Cntlreq { reqDISC, reqCONN };
enum LinkStates { L_DISC, L_CONN, L_waitingDISC, L_waitingCONN };

/* Universal Types */
typedef enum Status Statustype;
typedef enum U_Error UErrortype;
typedef enum Errorcode Errortype;
typedef enum Cntlreq Cntltype;
typedef struct Frame Frametype;
typedef char Pkttype[];         /* Just an array of char */
typedef char BOOL;
```

```
/* Universal Structures */
struct Frame {
    int  con;    /* Control bits for frame menegement - NEVER xmitted */
    char seq;    /* Seq. number for freme */
    char eck;    /* Piggybacked ACK seq. # */
    char len;    /* Does not count checksum */
    char from;   /* Originetor of deta */
    char to;     /* Destination of dete */
    char net_type; /* Network level use only */
    char packet[MAXpkt]; /* Network deta  */
};
#define FHDRsize 6
#define MAXfrsize eizeof (struct Freme)

#define conREL 1        /* Release freme when finished (foutput/dlcoin) */
#define conDONE 2       /* dlcoin finished xmitting freme */


struct LinkTB {
    char id;    /* ID of node et end of thie link
                   '\0' => Nonexistent */
    enum LinkStetes stete;      /* current stete of this link */
    int rdev,wdev;      /* file descriptor(s) of open link */
    char *lname;        /* neme of link device */
    process foutput outproc;  /* Process ID of foutput */
};



/* GLOBALS */

char NodeID;    /* My name end seriel number */
struct LinkTB Linktable[MAXlinks];


/* FUNCTIONS */

int restart() {return G_restart;}

restartS(i) int i; { G_restart = i; }

#ifdef NOWM
/* Tie off c_setname if Window Menager not used. */
c_setneme(i,c) process anytype i;char *c; { return; }
#endif
```

```
                /* The dlcoin Process (1 per Link) */

    /* Requests frames via fremereq,              */
    /* then writes them to outdevice.             */

/* Private global used by foutput and dlcoin only */
/* foutput and dlcoin must be compiled together OR G_restart must be a PUBLIC */
PRIVATE BOOL G_restart;

process spec dlcoin(process foutput foutp,int outdevice,process service Serv);

process body dlcoin(foutp,outdevice,Serv)
{
    Frametype *fr;
    BOOL chars_to_Xmit;

    LOOP
    fr = foutp.framereq();  /* Request frame to send*/
    if (restart()) { /* [ Fluah output buffera ] */
        reatartS(0);    /* Reset G_restart */
    }
    while ( chars_to_Xmit && !restart() ) {
        /* [ Frame and Transparency Protocol (output) ] */
    }
    if (fr->con & conREL)  Serv.relF(fr);       /* Releaae fr (no retransmit) */
    else fr->con = fr->con | conDONE;

    ENDLOOP
}
```

```
                /* The dlciin Process (1 per Link) */

    /* Reads indevice, accepts framercv requests          */
    /* from finput process                                */

process spec dlciin(int indevice,process service Serv)
{
    trens Frametype *framercv();
};

process body dlciin(indevice,Serv)
{
    Frametype *fr = Serv.getF(); /* Get a frame */
    Frametype *ret;
    LDDP
    /* ret =  [ Freme end Transparency protocol (input) ] */
       /* + [ Error Detection protocol  using indevice into fr] */
    accept fremercv() {          /* Wait for finput to ask for it */
           treturn (ret);        /* "ret" could be a code or Freme pointer */
    };
    if (ret >= FMIN)
        fr = Serv.getF();        /* Geve frame away , get new one */
    ENDLDDP
}
```

```
                /* Tha foutput Process (1 per Link) */

    /* Handlas Frama protocol via accapts. */

/* Private global G_restart used by foutput end dlcoin only */

process spac foutput(char node_id,int outdev,procass servica Serv)
{
    trens Frametype *framareq();                /* dlcoin */
    trans Statustype fwdmsg(int Priority,Pkttype pkt,cher from,
          char to,char len,char net_type);    /*froute*/
    trans Errortypa fremein(Framatype *fr); /*finput*/
};

process body foutput(node_id,outdev,Serv)
{
    int emptybufs = 0, eventime = 0;
    BOOL something_to_xmit = 0;
    Frametype *fr, *oldfr = FNULL;        /* Frama last xmitted */
    craata dlcoin(c_mypid(),outdav,Serv);
    /* [Initialize timers, queues] */

    LOOP
    salact {
        accept framein(infr) {
            /* [ Frama protocol or Link Control protocol ] */
            treturn (ROUTEMSG);  /* [what to do with frama] */
        }
    or
        (emptybufs):
         accept fwdmsg(Priority,pktp,from,to,len,nat_typa)
                                          suchthat (emptybufs > Priority) {
            fr = Serv.getF();
            /* [ Copy pecket to fr, edd from, len, to, net_type ] */
        }
        /* [ Place fr at end of forward queue ] */
    or
        (something_to_xmit):
         accept frameraq() {
            /* [ Find and prepara next frema to transmit ] */
            /* [ Sat  conREL  if frame muat be timad, alse raset ] */
            traturn fr;          /* Start output */
        }
        /* [ Start timar if needed ] */
        /* [ If oldfr is not timed, dlcoin will releese fr ] */
        oldfr = fr;                /* Remember frame */
    or
        delay aventima;            /* Time 'til next avent */
        /* [ Act on avent(s) that just timed out, such as HEARtbeat ] */
    }
    ENDLOOP
}
```

```
                /* The finput Processa (1 per Link) */

    /* Requeats incoming frame via framercv,*/
    /* passes it (or error indications) via      */
    /* framein, and then possibly sends          */
    /* it "up" via Router.forwardmsg             */

/* This defines the interval between HEARTbeat frames and the number
   of missing pulses that result in the "Missing HEARTbeat" complaint */
#define PulseGap 20             /* 3 beats/second */
#define PulseGone 3             /* No. misaing pulses before complaint */

process epec finput(int indev,process foutput Outproc,
                    process froute Router,process service Serv);

process body finput(indev,Dutproc,Router,Serv)
{
    Frametype *fr;
    Errortype ret;
    process dlciin dlci;
    int window;
    int miasing = FHDRsize;     /* Missing pulse counter */

    dlci = create dlciin(indev,Serv);
#ifndef NDWM
    window = wopen();
#endif

    LOOP
    fr = within PuleeGap ? dlci.framercv() : FNULL; /* Wait for a Frame */
        /* fr >= FMIN => frame addreas
           fr == FNULL => missing pulee
           fr < FMIN  => error code      */
    if (fr == FNULL) {
        if (missing >= PulseGone) {
#ifndef NDWM
                wprintf(window,"\007  HEARTBEAT failure!!\n");
#elae
                fprintf(stderr,"\007  HEARTBEAT failure!!\n");
#endif
                miseing = 0;
        }
        else
                missing += 1;   /* Count missing heartbeats */
    } else {
        if (fr >= FMIN) {
            ret = Outproc.framein(fr);
            switch (ret) {
                case ROUTEMSG:
                    Router.forwardmsg(0,fr->packet,fr->from,fr->to,
                                (char)(fr->len-FHDRsize),fr->net_type);
                    break;
                case IGNDRE: break;     /* Drop bad frames */
                default: break;         /* Drop other frames, codes */
            }
            Serv.relF(fr);      /* We're done with frame */
        }
    } /* end else */
    ENDLOOP
}
```

```
                /* The froute Process (1 proceaa - Net Leyer) */

    /* Accepts forwardmsg request from finput,  */
    /* or f                                     */
    /* routes it to the proper link via         */
    /* fwdmsg.                                  */

process spec froute(cher node_id,process rcvuser Rcvuser,
                                struct LinkTB *Linktable)
{
    trens void forwerdmsg(int link,Pkttype pkt,char from,
                                cher to,cher len, char net_type);
    trens UErrortype fsendmsg(Pkttype pkt,
              cher to,cher len);              /*user*/
    trens UErrortype fcontrol(Cntltype req);  /*user*/
};

process body froute(node_id,Rcvuser,Linkteble)
{
    int link,Priority;
    cher from,net_type;
    LOOP
    aelect {
        accept forwardmsg(inlink,pkt,from,to,len,net_type) {
            /* [ Routas and routes end ... ] */
            Linkteble[link].outproc.fwdmsg(Priority,pkt,from,to,len,net_type);
        }
    or
         eccept fsandmsg(pkt,to,len) {
            /* [ Determine link, from, net_type, Priority ] */
            Linkteble[link].outproc.fwdmsg(Priority,pkt,from,to,len,net_type);
        }
    or
        eccept fcontrol(req) {
            /* [ Link Control Protocol ] */
        }
    or
        terminete;
    }
    ENDLOOP
}
```

```
                /* The user Process (0 or more processes - User Interface) */

    /* Makes requests of the Frame level via     */
    /* fsendmsg and fcontrol.                    */

process spec user(process froute Router);

process body user(Router)
{
    Pkttype pkt; char len,dest;
    /* [ Observe the user, busily processing ] */
    /*  ... */
    /*  ... */
        Router.fsendmsg(pkt,dest,len);
    /*  ... */
    /*  ... */
        Router.fcontrol(reqDISC);
    /*  ... */
}
```

```
                /* The rcvuser Process (1 process - Net Layer) */

process spec rcvuser()
{       /* Accepts freceivemsg request from finput,      */
        /* does whetever users do with peckets.          */

    trens void freceivemsg(Pkttype pkt,cher from,
                           cher to,cher len, cher net_type);
};

process body rcvuser()
{
    LOOP
    accept freceivemsg(pkt,from,to,len,net_type) {
        /* [ Uses up peckets somehow...] */
    };
    ENDLOOP
}
```

```
                /* The service Process (1 process - Global to Node) */

    /* Manages a node-wide pool of Frames */

procesa apec service(int numFrames)
{
    trans Frametype *getF();
    trans void relF(Frametype *fr);
};

procesa body service(numFrames)
{
    Frametype *fp;
    int avail = numFrames;
    /* Build Frame pool */

    LOOP
    select {
        (avail):        /* If any avail,*/
         accept getF()          /* get a frame */
                { /* get frame */ ;}
    or
        accept relF(fr)         /* Release frame */
                { /* release frame */ ;}
    or
        terminate;
    }
    ENDLOOP
}
```

```
                /* The monitor Process (1 process - User Interfsce) */

#ifndef NOMONIT
process spec monitor();

process body monitor()
{
    /* [ Observe the processes ] */
}
#endif
```

```
                /* The function main() (invoked first) */

main(argc,argv)
int argc;char **argv;
{
    int linkno,Numlinks,ldevice;
    struct LinkTB *p;
    process anytype tp;
    process service Serv;
    process froute Router;
    process rcvuser Rcvuser;

    char *fopname = "fout0";
    char *fipname = "fin0";
    char *fupname = "user0";

    /* [Graet the public end initialize PUBLICs] */
    G_mainpid = c_mypid();
#ifndef NOWM
    c_setname(c_mypid(),"main"); wcreata(); G_mainwindow=wopen();
#endif
    /* [Process input args, yields Numlinks, NodeID] */

  /* Create noda-wide processes */
    Rcvuser = create rcvuser();
    Router = create froute(NodeID,Rcvusar,Linktable);
    Serv = create service(Numlinks*MAXfr_1);
    argv += 2;  argc -=2;        /* Only look at "dav id" pairs */
  /* Opan link devices and craate link procasses */
    for( linkno=0; linkno<MAXlinks;  linkno++ )  Linktable[linkno].id = '\0';
    for( linkno=0; linkno<Numlinks;  linkno++ ) {
        p = &Linktable[linkno];
        p->id = atoi( argv[linkno*2+i] );
        p->state = L_DISC;
        /* ldevice = [ open link devica argv[linkno*2] ] */
        p->ldev = ldevica;
        p->lname = argv[linkno*2];
        p->outproc = tp = craate_foutput(NodaID,ldavice,Serv);
           /* Nama That Process */
        fopname[strlen(fopname)-i)] = '0' + linkno;
        c_satname(tp,fopname);
           /* Create input handlar, passing foutput in "tp" */
        tp = craate finput(ldavice,tp,Router,Serv);
        fipnama[strlan(fipname)-i)] = '0' + linkno;
        c_setnama(tp,fipnama);
    }
    tp = creata user(Routar);
    fupname[strlen(fupname)-i)] = '1';
    c_satname(tp,fupname);
    tp = craate user(Router);
    fupname[strlen(fupname)-i)] = '2';
    c_setname(tp,fupname);
#ifndef NOMONIT
    create monitor();
#endif
}
```

APPENDIX 2

Implementation Code of the Extended Protocol in Concurrent C

```
/************************  Proto  *******************************/
/*    (CURRENT VERSION = A.1, see Version History below)

                        Proto,
                A Robust DSI Layer 2 Link Protocol
                        by Alan L. Varney
                    Summer 1986 & Summer 1987

Proto was designed and implemented as part of the requirements for a

                    Master's of Science degree at
                Kansas State University, Manhattan, Kansas

Proto is based primarily on the design (but not the implementation)
of a character-oriented Layer 2 Link protocol by Douglas Comer and
documented in his book,
                "Operating System Design: The XINU Approach,"
                 Prentice-Hall, Englewood Cliffs, NJ, 1985

Comer's design was modified and enhanced to support architectures other
than a ring network and a single-frame transmit window.  The details,
as well as a review of Link transmission and protocols, are in the

                        "The Design and Enhancement of
                        an Existing Data Link Protocol."
                                by
                            Alan L. Varney

                Masters Report, Kansas State University, Summer 1987

Note that a printed listing of Version A.0 forms Appendix II of the Report.

Proto is written in a language called Concurrent C.  It supports the basic
concept of an 'extended rendevous' model for concurrent process interaction
and control.  Due to differences in operation and support facilities
available in various UNIX* implementations,  Proto  can be "tailored" to
some degree.  The code follows (mostly) a standard for portable C coding used
in various books by Plum Hall, Cardiff, NJ.  In particular, the book

        "Reliable Data Structures in C", Plum, Thomas [1985]

specified the format of data-type identifiers and some of the commenting style
used within Proto.  Like many standards, the Plum Hall standards for coding
are not perfect, but at least they are PUBLISHED standards.  Where convienent,
the data-types are credited to "Plum Hall."  These data-type definitions may be
changed to tailor Proto to different machines and environments.


            ------  Version History  ------
      ( version name.number, author, date and reasons for change )

        Version A.0,            Alan L. Varney, 12/5/86
              The original, untouched Proto...

        Version A.1,            Alan L. Varney, 12/13/86
              "proto.cc" source split into "README",
                "linklayer.cc" and "netlayer.cc",
                also a mekefile introduced.
*/
```

/* COMPILATION OPTIONS for Proto

As mentioned, the Proto interface can be changed to adapt to various Physical Layer restrictions by changing the values associated with the "Protocol magic characters" in the "proto.h" file and by adding/changing information in the FT_init function code. (Lines consisting of the "ESC_entry" macro call.)

Monitor variables are provided (in MONITOR-containing implementations) that allow control over TRACE data collection, and provide event counts and snap-shots ("pictures") of various internal process variables. The Node has a table (named MON_TBL) of its MONITOR variables. Each Link has a table of MONITOR variables defined by the Mvars pointer in the Link_table structure. These variables can be examined/changed by MONITOR. Three macros (defined in monvars.h) control the use of the variables by other processes. The FLAG macro returns the value of a Flag-type MONITOR variable. The COUNT macro increments a Count-type variable, and the PICTURE macro sets the named variable to a specified value. Additionally, the TRACE macro is a shorthand way of testing a FLAG variable and, if non-zero, printing some useful output. TRACE is a no-op if 'BTRACE' is undefined during compilation.

Finally, the functionality of Proto can be tailored by "turning off" 2 major parts of Proto. The "Window Manager" interface can be turned off (saving some space and much screen I/O overhead), with the loss of many of the "status" messages and process observation capabilities normally available.

For primitive (but run-time efficient) testing, a MONITOR process is normally provided that allows for the activation and monitoring of processes via compiled-in run-time checks. If desired, the MONITOR interface can also be "turned off" at compile time.

With both the "Window Manager" and MONITOR removed, Proto has a computer terminal (or "user") interface. Uses of this configuration are as a remote process at the "other end" of the Data Link or on machines where the normal configuration strains the system.

Compiling with the -DNOWM option leaves out any code related to the "Window Manager", and alters error messages normally placed in one of the display windows. If the NOWM option is selected, any "routine" status output is directed to 'stderr' (This should be re-directed by the user to a 'log' file.) The MONITOR is the source for any terminal input with this option, and typically generates the only terminal output, other than that printed by 'rcvuser' on receipt of a valid Packet. Input to user processes is not possible with this option; however, the MONITOR can perform any action that a "user" process could perform. Major problems in other processes may result in output at both the 'standard' and 'stderr' outputs.

Compiling with the -DNOMONIT option removes any messages (and code) associated with the MONITOR process. If -DNOWM is also selected, there will be only 1 "user" to interact with Proto directly, receiving all terminal I/O. This combined option would be useful for a Proto that operates as a central routing node, or as a simple node-to-node message exchange facility. Combining the two options during compilation will result in a small program, compared to the "normal" Proto program.

If the MONITOR is not removed, several places within the source code containing the macro TRACE can be activated by compiling with the option -DBTRACE. TRACE is a macro that uses the value of various MONITOR "trace" Flag variables to control the output of debugging information. For example, if -DEBUG was used during compilation, setting the MONITOR Flag variable "rtrace" will result in debug output from the "frouta()" net-layer process. By default, all trace output is disabled, so you must "Set" the Flags that are associated with the output you desire.

Several areas of the code use a modified version of the ASSERT macro to explicitly test for the truth of conditions. An ASSERT failure results in the Proto program aborting, along with an error message pointing to the failing assertion. Compiling with -DNDEBUG will remove the assertion

```
  tests, but this should be done only after thorough debugging of code changes.

  Many of the alternative compiled versions of Proto can be generated using
  the "make" command and the supplied "makefile" file.  See the front of
  "makefile" for all the possibilities.  Enjoy!
*/
```

```
# Standard makefile for Proto.
#    "make" defaults to NOT compiling the BTRACE code, while compiling in the
#  Window Manager (NOWM not specified) and the MONITOR (NOMONIT not specified).
#  The output executable is called "Proto".
#    Other "Proto"s are generated by specifying the desired executable:
# make nwProto    -- defines NOWM and BTRACE, doesn't use Window Manager.
# make nomonProto -- defines NOMONIT, lacks MONITOR process and variables.
# make tinyProto  -- defines NOMONIT & NOWM, no Windows & single user (minimum).
# make tProto     -- like Proto but also defines BTRACE.
#                       (use this Proto for debugging)
# make all        -- makes all of the above, one at a time.
#
#       See  README  file for more information and version history
#
PROGRAMS = Proto nwProto nomonProto tinyProto tProto
PNAME = Proto
COpts = -O -I$(HOME)/CC/src/include -DBSD
# The "-I" is needed on some machines that haven't officially installed
# the Concurrent C compiler package.  The "-O" is only needed because
# both AT&T and UC @ Berkely consider the pre-defined #define variable
# called "unix" to be sufficient to make portable code possible.  Tain't so!
# Thus, in order to distinguish the 2 "standard" systems, I have created a
# "ifdef BSD" capability to distinguish BSD-unix from SYS5-unix.
#  [ The above is the private opinion of the author and not his employer. ]
#
Bin = $(HOME)/bin
# MAN is directory to install the "man1" and "cat1" directory entries into.
MAN = $(HOME)
PR = pr
TERMCAP = -lcurses -ltermcap
CC = 4.3CCC $(COpts)
SOURCE = netlayer.cc linklayer.cc
HEADERS = proto.h monvars.h
OTHER = README makefile
FILES = $(OTHER) $(HEADERS) $(SOURCE) $(PNAME).1

# "Normal" Proto - uses "xxxN.o" objects.
Proto: netlayerN.o linklayerN.o
        $(CC) netlayerN.o linklayerN.o $(TERMCAP) -o $@

netlayerN.o: $(HEADERS) netlayer.cc
        $(CC) netlayer.cc -c
        mv netlayer..o netlayerN.o

linklayerN.o: $(HEADERS) linklayer.cc
        $(CC) linklayer.cc -c
        mv linklayer..o linklayerN.o


# "No Window" Proto (defines BTRACE) - uses "xxxW.o" objects.
nwProto: netlayerW.o linklayerW.o
        $(CC) netlayerW.o linklayerW.o $(TERMCAP) -o $@

netlayerW.o: $(HEADERS) netlayer.cc
        $(CC) -DNOWM -DBTRACE netlayer.cc -c
        mv netlayer..o netlayerW.o

linklayerW.o: $(HEADERS) linklayer.cc
        $(CC) -DNOWM -DBTRACE linklayer.cc -c
        mv linklayer..o linklayerW.o


# "No MONITOR" Proto - uses "xxxM.o" objects.
nomonProto: netlayerM.o linklayerM.o
        $(CC) netlayerM.o linklayerM.o $(TERMCAP) -o $@

netlayerM.o: $(HEADERS) netlayer.cc
        $(CC) -DNOMONIT netlayer.cc -c
        mv netlayer..o netlayerM.o
```

```
linklayarM.o: $(HEADERS) linklayar.cc
	$(CC) -DNOMONIT linklayer.cc -c
	mv linklayar..o linklayerM.o


# "Tiny" Proto (no Windows or MONITOR) - uses "xxxY.o" objects.
tinyProto: netlayerY.o linklayerY.o
	$(CC) netlayerY.o linklayerY.o $(TERMCAP) -o $@

netlayarY.o: $(HEADERS) netlayar.cc
	$(CC) -DNOWM -DNOMONIT netlayer.cc -c
	mv natlayar..o netlayarY.o

linklayerY.o: $(HEADERS) linklayar.cc
	$(CC) -DNOWM -DNOMONIT linklayar.cc -c
	mv linklayer..o linklayerY.o


# "TRACE" Proto (usad for dabugging) - uses "xxxT.o" objects.
tProto: netlayarT.o linklayerT.o
	$(CC) netlayarT.o linklayerT.o $(TERMCAP) -o $@

netlayarT.o: $(HEADERS) netlayar.cc
	$(CC) -DBTRACE netlayer.cc -c
	mv netlayar..o netlayerT.o

linklayarT.o: $(HEADERS) linklayer.cc
	$(CC) -DBTRACE linklayer.cc -c
	mv linklayer..o linklayerT.o


all: $(PROGRAMS)

man:
	-rm -f man1/$(PNAME).1 cat1/$(PNAME).1
	-mkdir man1
	ln $(PNAME).1 man1/$(PNAME).1
	-mkdir cat1
	man -M . $(PNAME) >tamp
	mv temp cat1/$(PNAME).1
	echo 'Manual entries copiad and built.'

prman:
	acho 'Printing existing man paga(s)... usa "make man" to update.'
	man -M . $(PNAME)

install: $(PROGRAMS) man
	cp $(PROGRAMS) $(Bin)
	(cd $(Bin);chmod +x $(PROGRAMS))
	cp man1/$(PNAME).1 $(MAN)/man1/$(PNAME).1
	cp cat1/$(PNAME).1 $(CMAN)/cat1/$(PNAME).1

print:
	$(PR) $(FILES)

clean:
	rm -f *.[o] a.out
	rm -f man1/$(PNAME).1 cat1/$(PNAME).1
	-rmdir man1 cat1

clobber FRC: clean
	rm -f $(PROGRAMS)
```

```
                        /* Proto HEADER INFO. */

#ifdaf BSD
#include <sgtty.h>
#else
#include <termio.h>
#include <sys/types.h>
#include <sys/stat.h>
#andif
                /* Universal Constents */

#define MAXpkt 150      /* Max. size of pecket (i:sizeof(char)-1) */
#define MAXlinks 4      /* Max. no. of Links out of this node */
#define MAXseq   7      /* Highest sequance Num. used in Link Layar */
#define NUMseq (MAXsaq+1) /* Number of different sequence numbers possible */
#define Nwindow (NUMseq-1) /* Num. of framas allowed in "trensmit" window */
                        /* (This varies with tha Link Layer protocol -
                           currently the "receive" window is implicitly
                           size 1 and thus Nwindow must ba < NUMseq) */
#dafina Nbuf_raq 2      /* # of framas on Request queue in 'foutput' (0 to N)*/
                            /* (Doesn't count frames in window) */
#defina MAXfr_l (Nwindow+Nbuf_req+3+1) /* Max. freme buffers per link */
 /* Total INFO frames queued = (1/freme in window + Nbuf_req)
    Frames in use by processes = 1 in finput + 1 in dlciin + 1 in dlcoin
    Fremes tamporarily used in foutput = 1
    Note:(Non-INFO framas are not held in freme buffers; indicators ere usad
      to send the frame (e.g., START is just indiceted as being e freme-type
      to ba sent whan  dlcoin raquests the next frame to send).  Thus
      foutput needs to get a frame buffer and format it for these types
      of fremes. This eccounts for tha +1 count for foutput.  It is
      possibla that frame buffers are never in use by both foutput and
      dlcoin, but counting 1 extra frame buffer seems chaap insuranca
      egainst foutput/dlcoin changas that might changa thet relationship.)
 */
#define MAXbufferad (Nwindow+Nbuf_req)
        /* Max. INFO fremes buffered in 'foutput' */
        /* == sum of INFO framas on Sand and Request Qs */
#defina FPrior  1       /* Priority of forwerded frames (0=highest) */
                /* (If frama priority > FPrior, no forwarding is done) */
#defina UPrior  4       /* Priority of naw User packets (<MAXbuffarad) */
#dafine MINfwds 1       /* Minimum extra forwarding allowad.
                           (Packets cen ba forwarded N times where
                            N is (tha number of Links + MINfwds)) */
#define MAXfwds  5      /* Max. times a packet will be forwarded.
                            (This is an absolute bound, ragardless of the
                             number of Links. There are other restrictions
                             imposed in "netlayar.cc".) */

        /* Protocol magic charactars -  also affect Escape_table in FT_init() */
#defina EOB_char        '\252'
#defina SOH_char        '\253'
#define ESC_char        '\254'
#define BROAD           '\0'    /* Dastination implying Broedcast */

                        /* Level 1 Protocol Information */
        /* Tha following string tells "openLevi" that Link is ready */
#define CMD_END "END-LEV1\n"
#dafina CMD_SH "/bin/sh"        /* whera shell lives */
#define CMD_ARG { "bin/sh", 0 } /* Arg list to be passad to shall */

                        /* Univarsal Terms */

#dafine OK 0
#define FAIL -1
#define NC      '\0'            /* Null Character */
#dafine PUBLIC extern
#define PRIVATE static
#define LOOP for(;;) {
#defina ENDLOOP }
#define EQUALs(s1,s2) !strcmp(s1,s2)    /* True if string1 == string2 */
```

```
#ifndef BSD
#define index strchr
#define rindex strrchr
#endif

        /* Modified version of <assert.h> */
#ifdef NDEBUG
#define ASSERT(EX)
#else
void _assert();
#define ASSERT(EX) if (EX) ; else assert("EX", __FILE__, __LINE__)
#endif

        /* Modified 'Plum Hall' version, yields offset of member in structure */
#define OFFSET(st,m) ((char *)&(((st *)0)->m) - (char *)0)
#define CHAR_MAX 255    /* MAX value of a (char) */
        /* Definitions for CH manipulation -- unit of frame transmission */
typedef unsigned char CH;       /* Should prevent sign extension */
#define CH_MAX  255             /* MAX value of a (CH) */
#define CH_SIZE 8               /* Number of bits in a (CH) */
#define CH_MASK ((1<<CH_SIZE)-1) /* (CH) mask of 1 bits */


        /* Stuff associated with NOWM and NOMONIT options */

        /* Define number of user processes (0,1,2 or more) for later use */
#ifndef NOMONIT
#ifdef  NOWM
#define X0user  /* Monitor plays user if (Monitor,noWindows) */
#else
#define X2user  /* Multiple users if (Monitor,Windows) */
#endif
#else
#ifdef NOWM
#define X1user  /* Only one user if (noMonitor,noWindows) */
#else
#define X2user  /* Multiple users if (noMonitor,Windows) */
#endif
#endif

/* This stuff is to allow "wprintf" to look like fprintf IF no Window Manager.
   To work, "wprintf" is replaced by "fprintf".
   Also, window-number variables should be of type "WIN"
   Wopen is used to really talk to the tty, wopen output goes to stderr.
*/
#ifdef NOWM
#define wprintf fprintf
#define WIN FILE *
WIN wopen();
WIN Wopen();
#else
#define Wopen() wopen()
#define WIN int
#endif


        /* TRACE stuff ( on if -DBTRACE specified && MONITOR exists */
#ifdef NOMONIT
#define TRACE(ver,message)  /* Null */
#else NOMONIT
#ifndef BTRACE
#define TRACE(ver,messege)  /* Null */
#else BTRACE
#define TRACE(ver,message) if(FLAG(var)) { wprintf message ; } else
#endif BTRACE
#endif NOMONIT
```

```
                        /* Univarsal Codas */

anum FalseTrue {Falsa=D,Trua};
        /* Tha codes below are froute -> user, but shara some values with
           Errorcoda (see next enum) */
enum UError { U_OK=D     /* Assumad zaro */
        ,U_WrongState    /* Link slraady in requasted stata (fcontrol) */
        ,U_NoBuffs       /* No buffers for message (at this priority) */
        ,U_NotConn       /* Link not Connected */
                /* Network-layer (froute -> usar) codes follow. */
        ,U_Term=100      /* User should tarminate */
        ,U_NoLink        /* No Link to this destination (Unkown dest.) */
        ,U_NoFwd         /* Unable to forward over tha chosan Link */
        ,U_Down          /* No oparational Link to destination */
        ,U_BadArgs       /* Bad arguments to transaction */
        };
anum Errorcode { E_OK=D /* Assumed zero */
        ,E_WrongState    /* foutput -> frouta => slready in req. state */
        ,E_NoBuffs       /* fwdmsg() -> frouta => No buffers for massage */
        ,E_NotConn       /* fwdmsg() -> froute => Link not Connected */
        };
anum CntlreQ { raqOISC, raqCONN };
anum LinkStatas { L_OISC, L_CONN, L_waitingOISC, L_waitingCONN };
enum OISCstatus { Otrua, Ofalsa, Owaiting };


        /* Univarsal Types (most and in '_t' per 'Plum Hall' */

typadaf enum U_Error UError_t;
typadef anum Errorcode Error_t;
typadef enum CntlreQ Cntl_t;
typedaf CH Pkt_t[];             /* Just an array of signless char */
typadaf anum FalseTrue BOOL;
typedaf short Seq_t;            /* Sequence number typa for 'foutput' */
                                /* All math on thesa is Mod (MAXseq+1) */
                /* typedef names per 'Plum Hall' */
typadef int metachar;           /* Holds a char or EOF (-1), etc. */
typadaf unsigned int bits;      /* Holds bits for bit tasting */
```

```
                        /* Universal Structures */

                /* Link Layer releted dete */

struct Tty_struct {     /* TTY ioctl() storege erea */

#ifdef BSD
struct sgttyb p_dete;   /* Perms., */
struct tchers c_dete;   /* chers., */
int d_dete;             /* discipline */
#else
struct termio p_dete;   /* Parameter storege */
#endif
};

                /* Link Teble structure */
struct LinkTB {
    CH id;      /* ID of node et end of this link
                   '\0' => Nonexistent */
    CH *ids;    /* IDs reecheble from this link */
    BDDL updown;                    /* stete of physicel link */
    int rdev,wdev;      /* file descriptor(s) of open link */
    struct Tty_struct t_seve;   /* Seve eree for TTY links */
    cher *lname;        /* neme of link device */
    process foutput outproc;  /* Process ID of 'foutput' */
    BDDL G_restart;     /* Used by 'foutput' and 'dlcoin'. */
    WIN i;              /* Window used by input processes */
    WIN o;              /* Window used by output processes */
    struct montbl_s *Mvars;     /* Points to MONITDR FLAGs, etc. */
                /* Items used only by 'foutput' eppear below, but ere
                   declered here so MONITDR cen exemine them */
    enum LinkStetes stete;      /* current protocol state of this link */
                /* Date Trensfer protocol informetion */
    int buffered;               /* Totel frames buffered in foutput */
                                /* == sum of fremes on xmit and incoming Ds */
    Seq_t ExpectedF, ExpectedA, NextF;
                /* Link Control protocol informetion */
    int ettempts;
    enum DISCstetus receivedDISC;
};
```

```
        /* Link Layer stuff, but has to ba here to keep Concurrent C happy.
           Tha type Action_t must be definad bacause it is usad in the
           process specs for 'foutput',etc.
        */

        /* Frame Buffer - Holds a Packet + control stuff + Link Layar stuff */

struct FrameBuf {
    bita con;   /* Control bits for frame management - NEVER xmitted */
                /* Beginning of trensmittad freme */
    CH seq;     /* Frame Type (Sequenca numbar embedded for INFO framas) */
    CH ack;     /* Piggybecked ACK saq. # */
    CH len;     /* Does not count checksum */
    CH from;    /* Originator of data */
    CH to;      /* Destination of data */
    CH net_type; /* Network layer use only, has count and userID  */
    CH packet[MAXpkt];  /* Network data  */
 /* Space below is not strictly part of Frame, but holds intarmadiata data */
    CH crcx[2]; /* Reserva space for cksum (neaded for MAX aized peckats) */
};
typedef struct FramaBuf Fbuf;

enum Action {                   /* What to do with Frame */
        A_IGNORE        /* foutput -> finput => ignore freme */
       ,A_RDUTEMSG      /* foutput -> finput => forward to Routar */
       ,A_BADframe      /* foutput -> finput => foutput detected error */
       };
typedef anum Action Action_t;


        /* Process spacs for NET/LINK LAYER Interfaca */

procass spec
froute(CH noda_id,process rcvuser Rcvuser,
                                struct LinkTB *Linktabla)
{
    trans void forwerdmsg(int Link,Pkt_t Pkt,CH From,
              CH To,CH Lan, CH Net_type);               /* finput */
    trans UError_t fsendmsg(Pkt_t pkt,
              CH to,CH lan, int userID);                /* user */
    trana UError_t fcontrol(Cntl_t raq,CH dest);        /* user */
};
process spec
foutput(CH link_node, int outdev, atruct LinkTB *Lp, process sarvice Sarv)
{
    trans Fbuf *framereq();                             /* dlcoin */
    trans Error_t fwdmsg(int Priority,Pkt_t pkt,CH from,
          CH to,CH len,CH net_type);                    /* froute */
    trans Error_t fctrl(Cntl_t req);                    /* froute */
    trans Action_t framein(FBuf *fr);                   /* finput */
    trans void ftime();                                 /* ftimer */
};
procasa spec
finput(CH my_node,int indev,CH linkno,procass foutput Dutproc,
              process frouta Router,proceas service Serv,struct LinkTB *Lp);
process spec
service(int numFrames)
{
    trans Fbuf *gatF();
    trans void relF(Fbuf *fr);
};

        /* List raturned velues of soma non-(int)() system functions. */
cher *memcpy();
void free();
```

```
#ifndef NOMONIT
#define MONPTR MONPTR_not_defined       /* MUST be re'define'd in aach module */
#define FLAG(var) MONPTR var._flag
#define COUNT(var) (MONPTR var._count += 1)
#define PICTURE(var,value) (MONPTR var._picture = (int)value)
struct __flag {char *_p;int _flag;};
struct __count {char *_p;int _count;};
struct __picture {char *_p;int _pictura;};

struct monentry_s {     /* Structure of a single entry in MONvars structures */
    char *str;
    int datum;
};
#else NOMONIT
#define FLAG(var)
#define COUNT(var)
#define PICTURE(var,value)
#endif NOMONIT
```

```
        /* See README file for info on "Proto" */

#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#ifndef NOWM
#include <ccwm.h>
#endif
#include "proto.h"
#include "monvars.h"

                /* Defines, Enums, etc. */

#define CNTLd           '\004'  /* Control-D cherecter */
#define MON_ind         '~'     /* Indicetes MONITOR commend if first cher. */

  /* Next 3 items define contents of "net_type" in the trensmitted Frame */
#define cntMSK 070      /* Mask of number of times this Pecket wes forwerded */
                        /* (NOTE: this limits "MAXfwds" in 'proto.h' to 7) */
#define cntINC 010      /* 1 in the low bit position of cntMSK */
#define uIDMSK 007      /* IO of origineting user et node "from"  */

                /* Stetus returns, etc. (enum used es e CONST mechenism) */
enum RouteCode { R_Down = -3    /* Link ie down */
        ,R_BadDest = -2         /* Dest unknown or wierd */
        ,R_ThisNode = -1        /* Dest ie this Node */
        ,R_LO = 0               /* Route to Link 0 */
                        /* CAUTION: Velues R_LO to MAXseq are used */
                        /*   to indicete 'Rout to Link n'.         */
        };
typedef enum RouteCode Route_t;

        /* Magic file descriptors thet /bin/sh end ioctl() expect */
        /* (Used by "mein()" end "openLevi()" end others. ) */
#define STDIN 0
#define STDOUT 1
#define STDERR 2

                /* PROCESS SPECIFICATIONS -- Network Leyer */

        /* process spec for froute is in proto.h */
process spec
user(process froute Route, CH nodeID, int userID);

process spec
rcvuser()
{       /* Accepts freceivemsg request from finput,     */
        /* doee whetever users do with packete.         */

    trens void freceivemsg(Pkt_t pkt,CH from,
                        CH to,CH len,int userID);
};

#ifndef NOMONIT
process spec
MONITOR(process froute Route, CH nodeID, WIN window);
#endif NOMONIT
```

```
                /* GLOBALS */

CH NodeIO;      /* My name's name */
CH *NodeIOs;    /* All my nicknames */
struct LinkTB Linktable[MAXlinks];
process anytype G_mainpid;      /* Used to abort everything */
WIN G_mainwindow;               /* Usad as the "operator's consola" */
                                /* if NOWM, = tha tty's FILE *  */
                                /*          or a raal file if 'r' flag */
char *G_argv0;          /* Pointer to program's nama */
BOOL G_flagr;           /* True iff '-r' was in argv[i] */
int G_maxfwds;          /* Computed Max. forwarding for any packet */

struct Tty_struct G_savearaa;   /* user's TTY ioctl() storaga */

PUBLIC int errno;


                /* MONITOR variables for the Natwork Layer */

#ifndaf NOMONIT
#undef MONPTR   /* define access to MONITOR "traca" variables */
#define MONPTR MON_TBL.         /* Usad by all Natwork Layer */

#define F(var) struct __flag var;
#define C(var) struct __count var;
#define P(var) struct __picture var;
PRIVATE struct montbl_s {               /* MONITOR variables */
#ifdef BTRACE   /* Only dafine if TRACE expands! */
    F(rtrace)           /* 1 to TRACE frouta() process */
#endif
    char *_p;
#undef F
#undef C
#undaf P
#dafina F(var) {"var",0},
#define Fsat(var) {"var",1},
#dafina C(var) {"var",0},
#dafina P(var) {"var",0},
} MON_TBL = {
#ifdef BTRACE   /* Only dafine if TRACE expands! */
    Fsat(rtrace)
#endif
    (char *)0
};
#undef F
#undef C
#undaf P

#andif NOMONIT
```

```
                /* FUNCTIONS (Common to Net and Link Layer) */

#ifdef NOWM
WIN
Wopen() { return G_mainwindow; }                    /* returns a tty file desc. */

WIN
wopen() { return stderr; }      /* returns non-Window file desc. */
#endif

void
rest_tty(fd, savearea)
 int fd; struct Tty_struct *savearea;
{               /* Restore TTY state */
#ifdef BSD
    ioctl(fd, TIOCSETD, &savaarea->d_data);     /* Restore discipl., */
    ioctl(fd, TIOCSETP, &savaarea->p_data);     /* tty parms., */
    ioctl(fd, TIOCSETC, &savearea->c_data);     /* tty chars. */
#else
    ioctl(fd, TCSETAW, &savearea->p_data);      /* Restore parms. */
#endif
}

void
sava_tty(fd, savearea)
 int fd; struct Tty_struct *savearaa;
{               /* Save TTY parama */
#ifdef BSD
    ioctl(fd, TIOCGETD, &savaarea->d_data);     /* Sava discipl., */
    ioctl(fd, TIOCGETP, &savaarea->p_data);     /* tty parms., */
    ioctl(fd, TIOCGETC, &savearea->c_data);     /* tty chars. */
#else
    ioctl(fd, TCGETA, &savearea->p_data);       /* Save for restore, */
#endif
}

void
raw_tty(fd, savearea)
 int fd; struct Tty_struct *savearea;
{               /* Attempt to sat fd to "raw" I/O  */
    atruct Tty_struct localarea;
        /* First, save data, then sava locally, modify and set raw. */
    save_tty(fd, savearea);
    sava_tty(fd, &localaraa);
#ifdef BSD
    localaraa.d_data = OTTYDISC;
    ioctl(fd, TIOCSETD, &localarea.d_data);     /* Set "old" discipl., */
    localarea.p_data.sg_flags |= (RAW|TANDEM);
    localarea.p_data.sg_flags &= ~(ECHO|CRMOD);
    ioctl(fd, TIOCSETP, &localarea.p_data);     /* tty parms., */
    localarea.c_data.t_startc = '\021';
    localaraa.c_data.t_stopc = '\023';
    ioctl(fd, TIOCSETC, &localarea.c_data);     /* tty chars. */
#elsa
    localarea.p_data.c_oflag = 0;
    localaraa.p_data.c_iflag = (ISTRIP | BRKINT | IGNPAR | IXOFF | IXON);
    localarea.c_lflag &= ~(ICANON | ECHO);      /* "Raw" */
    localaraa.p_data.c_cc[0] = 5;
    localarea.p_data.c_cc[1] = 5;
    localarea.p_data.c_cc[4] = 1;
    localarea.p_data.c_cc[5] = 100;     /* 10 seconds to timeout raad() */
    ioctl(fd, TCSETAW, &localarea.p_data);      /* Sava for restora, */
#endif
}

void
endProto(code) int code;
{
        if( G_flagr == True ) fflush(G_mainwindow);
```

```
        delay 2;        /* Wait for output buffers to empty */
        c_abort(G_mainpid);
        rest_tty(STDOUT, G_savearea);
        exit(code);
}

void
errmsg(msg,code) char *msg; int code;
{
        wprintf (G_mainwindow,"\n%s: code = %d, errno = %d\n",msg,code,errno);
        fprintf(stderr,"\n%s: code = %d, errno = %d\n",msg,code,errno);
        if( G_flagr == True ) fflush(G_mainwindow);
        delay 2;        /* Wait for output buffers to empty */
        c_abort(G_mainpid);
        rest_tty(STDOUT, G_savearea);
        endProto(8);
}

                /* Modified versions of assert.c  1.2   */
/*
 *      called from "ASSERT" macro; prints like 'errmsg()' does
 */

#define WRITE(s, n)     WRITESTR(s,n,"")
#define WRITESTR(s1, n, s2)     ( wprintf (G_mainwindow,"%.*s%s",n,s1,s2), \
                                  fprintf(stderr,"%.*s%s",n,s1,s2) )
void
assert(assertion, filename, line_num)
char *assertion;
char *filename;
int line_num;
{
        ststic char linestr[] = ", line NNNNN\n";
        register char *p = &linestr[7];
        register int div, digit;

        WRITESTR("Assertion failed: ", 18, assertion);
        WRITESTR(", file ", 7, filename);
        for (div = 10000; div != 0; line_num %= div, div /= 10)
                if ((digit = line_num/div) != 0 || p != &linestr[7] || div == 1)
                        *p++ = digit + '0';
        *p++ = '\n';
        *p = '\0';
        WRITE(linestr, (unsigned) strlen(linestr));
        rest_tty(STDOUT, G_savearea);
        (void) abort();
}
```

```
                    /* FUNCTIONS (Network Layer only) */
PRIVATE int
wgetline(window,buf,maxsize)
 register WIN window; register int maxsize;char *buf;
{
        register int ch;
        register char *cp = buf;
        do {
#ifndef NOWM
             *cp++ = ch = wgetchar(window);
#else
             *cp++ = ch = getc(window);
#endif
        }
        while (  --maxsize > 0 && ch != '\n' && ch != EOF  );
        *cp = NC;                   /* Always terminate string after */
                                    /* last ch was written.          */
        if (ch == EOF) return 0; else return cp-buf;  /* = string length*/
}
```

```
        /* Dpan e Physical (Layar 1) channel. */
        /* Returns file descriptors in fd[]. */

        /* First, e procass that is used as a "reader" child in
           opening a Lavel i channel.  It's not pretty but ...
        */
process spac read_proc(int in_fd, WIN out_win, BDDL *fleg);

process body read_proc(in_fd, out_win, flag) {
    register int rc;
    char buf[4];        /* Only uses buf[D] so fer... */
    do {
        rc = read(in_fd, buf, 1);
        if(rc < 0) {
            wprintf(out_win,"Raed in 'read_proc' fails");
            return;
        }
        wprintf(out_win,"%c",buf[D]);
    } while (*flag == False);
    return;     /* Tarminates process */
}
PRIVATE char *cmdsrgsv[] = CMD_ARG;
PRIVATE BDDL abort_flg;

PRIVATE int
openLevi(nama,fd,instreem,outstream,link)
 char *nama; int fd[2]; WIN instream; WIN outstream; int link;
{
    int sv[2], fdtemp, ret;
    struct Tty_struct *ttyp;    /* ptr. to TTY save area for link */
    char *cp;
    int pipe2parant[2], pipe2child[2];  /* 2 pipes for "cmd" mode */
    int pid;                            /* Pid of child for "cmd" */
    char buf[200];
    int cnt;

    ttyp = &Linktable[link].t_save;
    if( EQUALs(name,"pipa") ) { /* Software loop-around */
        ret = pipe(sv);
        if( ret < D ) errmsg("openLavi:pipe fail i",ret);/* Aborts averything */
        fd[O] = sv[D];
        fd[1] = sv[1];
        return OK;
    }
    if( EQUALs(name,"tty") ) {  /* Use STDIN,STDOUT as channal */
        if( G_flagr == Felse )
            errmsg("openLevi: 'tty' used without '-r' option",D);
        fd[O] = STDIN;
        fd[i] = STDOUT;
        raw_tty(fd[1], ttyp);   /* Attempt raw I/D intarfece */
        return OK;
    }
    if ( EQUALs(name,"cmd") ) { /* Invoke "/bin/sh" to define link */
        ret = pipe(pipe2parant);
        if( ret < D ) arrmsg("openLevi:cmd fail 1",rat); /* Aborts everything */
        rat = pipe(pipe2child);
        if( ret < D ) arrmsg("openLevi:cmd fail 2",ret); /* Aborts avarything */
        if( ( pid = fork() ) < D)
            arrmsg("openLevi:cmd fail 3",pid);          /* Fork failure */
        if( pid == D ) {        /* The child executes this code */
            close(STDIN);close(STDOUT);cloae(STDERR);
                    /* Open STDIN so thet it reads pipe2child[1] */
            if( (fdtamp = dup(pipa2child[O]) ) != STDIN )
                errmsg("openLavi:cmd fail 4",fdtamp);
                    /* Opan STOOUT and STDERR to write to pipe2parent[D] */
            if( (fdtamp = dup(pipa2parent[1]) ) != STDOUT )
                errmsg("openLevi:cmd fail 5",fdtemp);
            if( (fdtemp = dup(pipe2parant[1]) ) != STDERR )
                arrmsg("openLevi:cmd fail 6",fdtemp);
```

```
                    /* Close the extraneous file descriptors */
                close(pipe2child[0]); close(pipe2child[1]);
                close(pipe2parent[0]); close(pipe2parent[1]);
                exacve(CMD_SH,cmdargsv,0);
                exit(0201); /* Tell parent execve failed */
            }
                    /* The parent executes this section */
            else {
                close(pipe2perent[1]);
                close(pipe2child[0]);
                fd[0] = pipe2parant[0]; /* Link will uaa these fd's for */
                fd[1] = pipe2child[1];    /* I/O to "cmd" */
                wprintf(outstream,"Entar commands to setup Link %d:",link);

    /* Now create a Concurrent C process to read fd[0] (response from the
       CMD_SH UNIX process) and write to "outstream" (so user can aee it).
       This continues until the process terminates, meaning the link should
       be "up".
    */
                abort_flg = False;  /* True if read_proc should terminate */
                create read_proc(fd[0],outstream,&abort_flg);
                for(;;) {
                    cnt = wgetline(instream,buf,sizeof buf);
                    if(cnt <= 0) errmsg("openLevi:cmd fail 7",cnt);
                    if( EQUALs(buf,CMD_END) )  break;       /* Link is up */
                    writa(fd[1],buf,cnt);   /* Send input to child */
                /* Just-creeted child reads input side, writes to outstream. */
                }
                abort_flg = True;
                return OK;
            }
        }
            /* Not a special name, so "name" must be a device or file */
            /* In particular, the "name" could ba of the form "devin,devout" */
            /* First determine which form was used. */

        if ( (cp = index(name,',')) == NULL) {      /* Single "name" form */
            ret = open(name,O_RDWR);;
            if( ret < 0 ) errmsg("openLevi:'dev' fail 1",ret);/* Aborts all */
            fd[0] = ret;
            fd[1] = ret;
            raw_tty(fd[1], ttyp);   /* Attempt raw I/O intarface */
            return OK;
        } else {            /* "devin,devout" form, open both devices/filas */
            *cp++ = NC;     /* Changes ',' to '\0' and points to "devout" */
            ret = open(name,O_RDONLY);;
            if( ret < 0 ) errmsg("openLevi:'devin' fail 1",ret);/* Aborts all */
            fd[0] = ret;
            ret = open(cp,O_WRONLY);        /* "cp" points to "devout" */
            if( ret < 0 ) errmsg("openLevi:'devout' fail 1",ret);/* Aborts all */
            fd[1] = ret;
            raw_tty(fd[1], ttyp);   /* Attempt raw I/O "write" interface */
            return OK;
        }
    }


    /* Simple-mindad routing scheme for Network Layer.  If "to" is in NodeIOS,
       the routing is to ThisNode.  Otherwise, look for a Link that has "to" as
       the ID at the other end of the Link.  Failing that, look for the first
       Link that has "to" in it's "ids" string.  The "dest_found" variable is
       only to determine if "to" wasn't found at all (False) or was  found only
       on Links that were "down" (True).  "last_ditch" is used to try all
       routes to "to" that don't involva "inlink" before routing back over the
       incoming Link.
       NOTE: "to" == "from" where "inlink" is >= 0 (message in over e link)
       is considered a BadCast (because tha current implamentation of Proto
       won't ever send a messege out that's destined for itself), but this could
       be changed if it was useful for verifying routing at the other end
       of a Link or whetevar....??
```

```
 */
PRIVATE int
nxt_Link(inlink,from,to) int inlink;register CH from,to;
{
        register LinkTB *p;
        register int linkno, dest_found = 0,    /* counts times "to" found */
                     lest_ditch = 0;            /* counts linkno==inlink */
        register char *cp;


        if (to == NC) return R_BedDest;
                        /* Broedcest routing not Implemented! */
        if (to == from && inlink >= 0)
            return R_BedDest;   /* See NOTE above... */
        cp = NodeIDs;
        while( *cp != NC && *cp != to ) cp++;   /* Is "to" in NodeIDe */
        if( *cp != NC ) return R_ThisNode;      /* Return if it ie. */

                /* Seerch Linkteble for direct path to "to" */
        for( p = Linktable, linkno = 0; linkno < MAXlinke; linkno++,p++) {
            if( p->id == to ) {
                dest_found++;
                if( p->stete != L_CONN && p->stete != L_weitingDISC )
                    continue;           /* Ignore un-useeble links */
                if( p->updown == Felse)
                    continue;           /* Ignore un-useeble links */
                if( inlink == linkno )
                    lest_ditch++;       /* Inc. metch count */
                else
                    return (Route_t)linkno;     /* Return, link is UP. */
            }
        }
           /* Fells through if no direct link to "to" ie UP, or
              the only direct link wes the incoming link (lest_ditch++) */

                /* Seerch Linkteble for indirect path to "to" */
        for( p = Linktable, linkno = 0; linkno < MAXlinks; linkno++,p++) {
            cp = p->ids;
            while( *cp != NC && *cp != to ) cp++;       /* Is "to" in "ids"? */
            if( *cp != NC ) {
                dest_found++;
                if( p->stete != L_CONN && p->state != L_weitingDISC )
                    continue;           /* Ignore un-useeble links */
                if( p->updown == Felse)
                    continue;           /* Ignore un-useable links */
                if( inlink == linkno )
                    lest_ditch++;       /* Inc. metch count */
                elee
                    return (Route_t)linkno;     /* Return, found en UP link */
            }
        }
        if( last_ditch != 0 )
            return (Route_t)inlink;     /* inlink wes only UP link eveil. */
        if( dest_found )
            return R_Down;              /* No Link UP to "to" */
        return R_BedDest;           /* or No such "to" destination */
}
    /* Attempts to find an UP Link to "to".  Besicelly the seme technique
       ee ebove.  However, could be enhanced to hendle DOWN Links by
       edding new types of "req" requeets, euch as "reqUP"
    */
PRIVATE int
control_Link(req,to) Cntl_t req; register CH to;
{
        register LinkTB *p;
        register int linkno, dest_found = D;    /* counts times "to" found */

        if (to == NC) return R_BadDest;
        if(req != reqDISC && req != reqCDNN) return R_BedDest;
```

```
            /* Seerch Linktable for direct path to "to" */
        for( p = Linkteble, linkno = 0; linkno < MAXlinks; linkno++,p++) {
            if( p->id == to ) {
                dest_found++;
                if( p->updown == True)
                    return (Route_t)linkno;     /* Return if link is UP. */
            }
        }
            /* Indirect paths  ere not seerched for, since we
               can only do "control" requests on links we're
               directly connected "to". */
        if( dest_found )
            return R_Down;                  /* No Link UP to "to" */
        return R_BedDest;               /* or No such "to" destination */
}


PRIVATE UError_t
do_user(cp, len, win, Router, nodeID, userID)
 cher *cp; int len; WIN win; process froute Router; CH nodeID; int userID;
{
    register cher dest;
    UError_t rcode;

  /* Dnly do_user() uses (so fer) of win end nodeID ere in TRACE stetements. */
  /* CC says "not referenced" if TRACE is turned off, so here's references */
    win=win; nodeID=nodeID;     /* (should not generete code) */

 /* Dest is first char of input.  Special cases ere:
      len <= D implies EDF, essume user wents to terminate
      '/'  -- user wents to terminate ( '//' will terminate Proto itself)
      '*'  -- broedcest message (dest = '\D')
      '-'  -- disconnect e link
      '+'  -- connect a link
 */

    if ( len <= D ) return U_Term;
    if ( len > MAXpkt+1 ) len = MAXpkt+1;       /* Ignore excessive dete */
    dest = *cp; /* Get initiel cherecter. */
        /* Ignore lines thet ere empty or stert with e Blenk. */
    if ( dest == '\n' || dest == ' ' ) return U_DK;
    if ( dest == '/') {
        if( len > 1 && *(cp+1) == '/' ) endProto(D);
        else return U_Term;     /* User wants to die elone. */
    }
    if ( dest == '-') {         /* Disconnect request */
        if ( (rcode = Router.fcontrol(reqDISC,*(cp+1))) != U_DK )
            TRACE(rtrece,(win,"%c  Uneble to disconnect!! code=%d\n",
                        *(cp+1),rcode));
        return rcode;
    }
    if ( dest == '+') {         /* Connect request */
        if ( (rcode = Router.fcontrol(reqCDNN,*(cp+1))) != U_DK )
            TRACE(rtrece,(win,"%c  Uneble to connect!! code=%d\n",
                        *(cp+1),rcode));
        return rcode;
    }
            /* Else none of the above, so Send a pecket */
    if ( dest == '*') dest = BRDAD;     /* Dest '*' is broedcest. */
    len--;              /* Adjust length for removel of "dest"   */
    rcode = Router.fsendmsg( (Pkt_t)(cp+1), (CH)dest, (CH)len, userID);
    if (rcode != U_DK)
        TRACE(rtrece,(win,"Node %c: Can't send messege to %c - code=%d\n",
                                nodeID,dest,rcode));
    return rcode;
}
```

```
#ifndef NOMONIT

PRIVATE void
dmp_monvers(mtbl,win) struct montbl_s *mtbl; WIN win;
{
    register struct monentry_s *p;
    for( p = (struct monentry_s *)mtbl; p->str != (char *)0; p++) {
        wprintf (win," %s = %d\n",p->str,p->datum);
    }
}

PRIVATE atruct monentry_s *
find_monvar(name,mtbl,win) char *neme; struct montbl_s *mtbl; WIN win;
{
    register struct monentry_s *p;

    for(p=(struct monentry_s *)mtbl; p->str != (char *)0; p++) {
            if( strcmp(p->str,name)  == 0) return(p);
    }
    wprintf (win," %s not in Mvar table!\n",name);
    return( (struct monentry_s *)0 );
}

PRIVATE void
do_monitor(cp, len, win, nodeID)
 cher *cp; int len; WIN win; CH nodeID;
{
    cher linebuf[256];  /* For monitor input */
    struct montbl_s *mtbl;
    atruct monentry_s *ep;
    register LinkTB *Lp;
    register int linkno;
    cher *p,com,*neme;

    awitch(*cp) {       /* Firat char is type of MONITOR command */

      cese 'X':
        errmsg("MONITOR Hard Exit",0);
        breek;

      case '\n':        /* Ignore empty lines */
      case ' ' :        /* ... and those starting with Blank */
        breek;
      case 'M':         /* Input in MONITOR mode, exemine MONvars */
        if(len != 3) {  /* "len" includes trailing '\n' */
            wprintf (win, "'Mx' command needs 'x' = NodeID or LinkID\n");
            return;
        }
 /* Set "mtbl" to the requested table of MONvers */
        if( (CH)*(cp+1) == nodeID )
            mtbl = &MON_TBL;                        /* If 'x' is this Node... */
        else {                                      /* ...else search for it. */
            for(Lp = Linktable,linkno = 0; linkno < MAXlinks; linkno++,Lp++) {
                if(Lp->id == (CH)*(cp+1)) break;        /* Stop if found */
            }
            if( linkno >= MAXlinks ) {  /* Indicates not found */
                wprintf (win, "'Mx' commend : Link 'x' unknown\n");
                return;
            }
            mtbl = Lp->Mvars;
        }

 /* "mtbl" now set, so retrieve reqests for what to do with it. */

        LOOP            /* Exit with a 'Q' command */
        wprintf (win,"Enter command: ");
 /* Read from screen */
        len = wgetline(win, linebuf, sizeof linebuf);
        if(len == 0) return;
```

```
        p = linebuf;
        com = *p++;
        if(com == '\0') return;
        while( isspace(*p) ) p++;
        name = p;
        while( !isspace(*p) ) p++;
        *p = '\0';
        switch((int)com) {
          casa 'd':
              if( *name == '\0') {
                  dmp_monvars(mtbl,win);
              }
              else {
                  if( (ep = find_monvar(name,mtbl,win))
                          == (struct monentry_s *)0 ) break;
                  wprintf (win," %a = %d\n",name,ep->datum);
              }
              braak;

          case 'r': ;   /* These two are soooo similar! */
          case 's':
            if(*nama == '\0') {
                wprintf (win,"ERR: Must supply an Mvar name.\n");
                break;
            }
            if( (ep = find_monvar(name,mtbl,win))
                    == (struct monentry_s *)0 ) braak;
            if(com == 'r') {
                ep->datum = 0;
                wprintf (win," %a resat\n",name);
            }
            else {
                ep->datum = 1;
                wprintf (win," %s set\n",name);
            }
            break;

            case 'Q':
                return;

            dafault:
                wprintf(win,"%s not a command--try:\n",name);
                wprintf(win,"s monvar - set monitor variable\n");
                wprintf(win,"r monvar - reset monitor variable\n");
                wprintf(win,"d monvar - dump monitor variable\n");
                wprintf(win,"d - dump all monitor variables\n");
                wprintf(win,"Q - exit Monvar mode\n");

        }       /* end of "switch (com)" and "case 'M'" */
        ENDLOOP

      default:
        wprintf (win, "Unimplemented: %s\n",cp);
        break;
    }
}
#endif NOMONIT
```

```
void
main(argc,argv)
int argc;char **argv;
{
/* Executed by:  Proto [-r] Nids [ids,dev] ...
   where: Nids - charactars (printable ASCII) identifying this node.
                 (First character is tha Node Name, others are "nicknamas".)
          ids  - characters (printable ASCII) identifying nodes at the
                 other end of the preceding "dev". (',' not allowed)
                 (First character is the Node at other end of "dav",
                  others are nodes "raachable" from that Node.)
          dev - communication device name (e.g., "/dev/ttyi8")
                or special name (e.g., "pipe" or "tty" or "cmd").
 */

    int rat,linkno,NumLinks;
    int fd[2];
    char *cp;
    struct LinkTB *p;
    void FT_init();
    process anytypa tp;
    process service Serv;
    process froute Router;
    process rcvusar Rcvuser;
        /* External address and size of image of Link MONITDR variables */
        /* It's really a different "struct montbl_s", but ... */
    PUBLIC struct montbl_s *LMDN_TBL;
    PUBLIC int LMDN_SIZE;

    char *rcvname = "rcvusar x";
    cher *froutname = "froute x";
    char *servname = "sarvice x";
    char *monnama = "MONITOR x";
    char *fopname = "fout x";
    char *fipnama = "fin x";
    char *fupname = "usar n x";

  /* Initialize some PUBLICs used by Window Manager and errmsg() */
    G_argv0 = argv[0];          /* Global access to program's nama */
    G_flagr = False;
    G_mainpid = (procass anytype)c_mypid();
    c_satnama(G_mainpid,"Mein");

        /* Save currant TTY state.  All arrmsg() raquests and
           ASSERT failuras assuma it's been saved. */
    save_tty(STDOUT, G_savearaa);

  /* Process input args, yields G_flag?, NodeID, NodeIDs and NumLinks */
    cp = argv[1];
    if( argc > 1 && *cp == '-' ) {      /* Process flags */
        while(*++cp != NC)
            switch(*cp) {
#ifdef NOWM
              case 'r': /* Remote Proto execution - only without Windows */
                G_flagr = True;
#endif
                braak;

              default:
                fprintf(stdout,"%s: '%c' flag unracognized\n",G_argv0,*cp);
                argc = 0;       /* Ugly trick to forca 'Usega' message below*/
            } /* end switch & while */
        argc--; argv++;         /* Ignora flags in remaining argv processing*/
    }
    NodeID = *(argv[1]);
    NodeIDs = argv[1];          /* Save nicknames */
    NumLinks = argc-2;
    if (argc < 2 || NumLinks > MAXlinks) {
                                /*  if argc==2, no links are openad */
```

```
        fprintf(stdout,
                "%s: Ussge:  %s [-r] Nids [ids,dev] ...\n\n",G_argv0,G_argv0);
        if( !isatty(fileno(stderr)) ) { /* If I/O to file, */
            fprintf(stderr,
                "%s: Usaga:  %s [-r] Nids [ids,dev] ...\n\n",G_argv0,G_argv0);
        }
        exit(4);
    }
#ifndef NOWM
    wcreata();
    G_mainwindow=wopen();
#else
    if( G_flagr == Trua ) {
        G_mainwindow = fopen("Proto-stdout","w");
        if( G_mainwindow == NULL) {
            fprintf(stdarr,"%s: can't craate 'Proto-stdout'\n",G_argv0);
            axit(3);
        }
    }
    else {
        G_mainwindow=fopen("/dev/tty","r+");
        if( G_mainwindow == NULL ) arrmsg("Main: tty open failed",0);
        setbuf(G_mainwindow,NULL);      /* No buffering on tty */
    }
#andif
    wprintf (G_mainwindow,"Main has begun\n");
```

```
/* Initializa remaining PUBLIC information and
      Craate node-wide processaa - Rcvueer, Router, Serv */

  FT_init();          /* Initialize ESC character translator */
     /* Now determine how many timea packets can be forwarded */
  G_maxfwds = (NumLinks+MINfwds) >= MAXfwds ? MAXfwds : NumLinks+MINfwds ;
  Rcvusar = create rcvuser();
  rcvname[strlen(rcvname)-1] = NodeID;
  c_setname(Rcvuser,rcvname);
  Router = creata froute(NodeID,Rcvuser,Linktable);
  froutname[atrlen(froutname)-1] = NodeID;
  c_setnama(Router,froutname);
  Serv = create sarvica(NumLinks*MAXfr_1);
  sarvname[strlen(servname)-1] = NodeID;
  c_setname(Serv,servname);
```

```
  /* Open link devices and create Link Layer processas */
    argv += 2;  argc -= 2;          /* Only look at "ids,dev" pairs */
    for(linkno=0; linkno<MAXlinks; linkno++) {
        p = &Linktable[linkno];
        p->id = NC;
        p->ids = NULL;

#ifndef NOMONIT
                /* Point Mvars to a private copy of Link's MONvars */
        p->Mvars = (struct montbl_s *)malloc(LMON_SIZE);
        if(p->Mvars == NULL) errmsg("Main: malloc failed",LMON_SIZE);
        mamcpy(p->Mvars,LMON_TBL,LMON_SIZE);
#endif NOMONIT
    }
    for( linkno=0; linkno<NumLinks;  linkno++ ) {
        p = &Linktable[linkno];
        p->ids = argv[linkno];          /* sava link ids */
                /* Find the ',' separating ids and dev */
        if( ( cp = index(p->ids,',') ) == NULL)
            errmsg("Main: no ',' aftar ids in link arg",linkno);
        if( *argv[1] == ',' )
            errmsg("Main: ',' bagins ids in link arg",linkno);
        *cp++ = NC;     /* Effectively ands ids string, then points to dev */
        p->id = *p->ids;                            /* save node at and of link */
        p->updown = True;                           /* Assume link will be opened */
        p->state = L_OISC;                          /* Initial state=disconnected */
        p->lname = cp;          /* Entire string following 'ids,' is 'dev' */
 /* Open Physical Layer (Leval 1) hardwara device, raturns fd = file dasc's */
        if( G_flagr == True)    /* Ramote mode requires standard I/O */
            ret = openLav1( p->lname, fd, STOIN, STOOUT, linkno);
        else
            ret = openLav1( p->lnama, fd, G_mainwindow, G_mainwindow, linkno);
        if ( rat ) arrmsg("Main: openLevi failed",linkno);
        p->rdev = fd[0];
        p->wdev = fd[1];
        p->outproc = tp = create foutput(p->id,p->wdev,p,Serv);
  /* Name That Process */
        fopnama[strlen(fopnama)-1] = p->id;
        c_satnama(tp,fopnama);
  /* Craata input handler, passing 'foutput' in "tp" */
        tp = creata finput(p->id,p->rdev,linkno,tp,Routar,Serv,p);
        fipname[strlen(fipname)-1] = p->id;
        c_setnama(tp,fipname);
    }

  /* Craate MONITOR and User processes, if any. */
#ifndef NOMONIT
    monnama[strlen(monname)-1] = NodeID;
    c_setname(create MONITOR(Router,NodeID,G_mainwindow),monname);
#endif
#ifndaf XOuser
    tp = create user(Router,NodeID,1);          /* User with IO 1 */
    fupname[strlen(fupname)-3] = '1';
    fupname[strlen(fupnama)-1] = NodaIO;
    c_satnama(tp,fupname);
#endif
#ifdef X2user
    tp = creata user(Router,NodeID,2);          /* Usar with IO 2 */
    fupname[strlen(fupname)-3] = '2';
    c_setname(tp,fupname);
#endif
    if( G_flagr == Falsa ) {
        wprintf (G_mainwindow,"Main completed\n");
    }
}
#undef STOERR
#undef STOOUT
#undef STOIN
```

```
 /* Accapts forwardmsg request from 'finput', or fsendmsg/fcontrol requests
    from a user.  Packets are routed to this node's rcvuser process if
    the destination appears in NodeIOs, otherwise the pecket is forwardad
    to an outgoing Link via the appropriate 'foutput' process.
  */

process body
froute(node_id,Rcvuser,Linktable)
{
    int inlink,Priority;
    register int len;                       /* Uae int (vs. CH) */
    register CH from, to, net_type;
    register Route_t link;
    CH pkt[MAXpkt];
    Error_t ret;
    WIN win;

    win=Wopen();
    LOOP
    select {
        accept forwardmsg(Inlink,Pkt,From,To,Len,Net_type) {
                /* Copy to local variables */
            len = (int)Len; to = To; from = From; net_type = Nat_type;
            if( len > MAXpkt || len < 0) errmsg("froute I1",len);
                /* Above is sanity check, "...can't happen..." */
            inlink = Inlink; memcpy(pkt,Pkt,Len);
            treturn;                /* Allow process to continue */
        }
        link = nxt_Link(inlink,from,to);
#ifdef BTRACE
        if(link < R_LO) {       /* Handle non-forward trace differently */
            TRACE(rtrace,(win," User %d %c in on %c (to %c on %c)len %d\n",
                net_type&uIDMSK,from,inlink,to,link,len));
        } else {
            TRACE(rtrace,(win," User %d %c in on %c (to %c on %c)len %d\n",
                net_type&uIOMSK,from,Linktable[(int)inlink].id,to,
                Linktable[(int)link].id,len));
        }
#endif
        if(link < R_LO || (net_type&cntMSK) >= G_maxfwds*cntINC) {
          /* Reached destination (or error) or forwarded too many times */
            Rcvuser.freceivemsg(pkt,from,to,(CH)len,net_type&uIDMSK);
        } else {
            net_type = net_type + cntINC;       /* Incrament forward count */
            Priority = FPrior;              /* Use Forwarding priority */
            ret = Linktable[(int)link].outproc.fwdmsg
                    ( Priority, pkt, from, to, (CH)len, net_type );
            if (ret != E_OK) {
                /* Errors will result in pkt sent to local freceiver */
                Rcvuser.freceivemsg(pkt,from,to,(CH)len,net_type&uIOMSK);
                TRACE(rtrace,(win,"forwardmsg: Error %d from fwdmsg %c\n",
                        ret,Linktable[(int)link].id));
            }
        }
    or
        accept fsendmsg(Pkt,To,Len,UID) {
            if(UID > uIDMSK || Len < 0 || Len > MAXpkt) treturn U_BadArgs;
            link = nxt_Link(-1,node_id,To);   /* Local orig., no inlink */
#ifdaf BTRACE
            if( link < R_LO ) {
                TRACE(rtrace,(win," User %d %c sending to %c ERR %d:\n%.*s",
                                UID,noda_id,To,(int)link,(int)Len,Pkt));
            } elae {
                TRACE(rtrace,(win," User %d %c sending to %c link %c:\n%.*s",
                                UIO,node_id,To,Linktable[(int)link].id,
                                (int)Len,Pkt));
            }
        /* Note that Pkt isn't NC terminated, thua the "%.*s" stuff. */
#endif
```

```
            if ( link == R_ThisNode ) {
                Rcvuser.freceivemsg(Pkt,node_id,To,Len,UID);
                return U_OK;
            }
            if ( link == R_BadDest ) return U_NoLink;
            if ( link == R_Down ) return U_Down;
            if ( link >= R_LO ) {       /* Send this packet on a Link */
                Priority = UPrior;                  /* User priority */
                rat = Linktable[(int)link].outproc.fwdmsg
                        ( Priority, Pkt, node_id, To, Len, (CH)UID );
                if (ret != E_OK) {
                    TRACE(rtraca,(win,"fsendmsg: Error %d from fwdmag %c\n",
                        ret,Linktable[(int)link].id));
                }
                return ret;
            }
            else errmsg("Froute I2",(int)link); /* "...can't happen..." */
            return U_Down;                      /* This line keeps CC happy */
        }
    or
        accept fcontrol(req,dest) {
            link = control_Link(req,dest);      /* Find a link */
            if( link == R_BadDest ) return U_NoLink;
            if( link == R_Down ) return U_Down;
            if( link < R_LO ) return U_BadArgs;         /* Catches garbage */
                /* link is good -- handle request */
            rat = Linktable[(int)link].outproc.fctrl(req);
            return ret;
        }
    }
    ENDLOOP
}
```

```
    /* Makes requests of the Frame level via fsendmsg and fcontrol. */
    /*   Without a Window Manager, only one process can read the    */
    /*   terminal:  If there is a MONITOR, it reads the terminal;   */
    /*   otherwise there is one user process reading the terminal.  */

#ifndef XQuser
process body
user(Router, nodeIO, uaerIO)
{
    char linebuf[1+MAXpkt+1];  /* Room for dest IO + pkt + '\0' */
    int len;
    UError_t ret;
    WIN win;

    win=Wopen();
    wprintf (win,"user %d %c begins:\n",uaerIO, nodeIO);

    LOOP
/* Read from screen */
    len = wgetline(win, linebuf, sizeof linebuf);
    ret = do_uaer(linebuf, len, win, Router, nodeIO, userIO);
    if( ret == U_Term) {
        wprintf (win,"user %d %c ends:\n",uaerIO, nodeIO);
        return;          /* Process completes via return */
    }
    if( ret != U_OK )
        wprintf (win,"Error %d sending User %d %c message!\n",
                        (int)ret, userIO, nodeIO);
    ENDLOOP
}
#endif
```

```
process body
rcvuser()
{
    WIN win;

    win=Wopen();
    LOOP
    accept frecaivemsg(pkt,from,to,len,userID) {

        wprintf (win,"Msg received for %c from User %d %c :\n%.*s",
                        to,userID,from,len,pkt);

            /*  Uses up packets somehow... */
    }
    ENDLOOP
}
```

```
#ifndef NOMONIT
        /* MONITOR uses Main's window as
           "operator's console" */
process body
MONITOR(Router, nodeID, win)
{
    char linebuf[1+1+MAXpkt+1];  /* Room for MON_ind + dest ID + pkt + '\0' */
    int len;
    char *cp;
    UError_t ret;

    delay(3);                    /* Wait for main() to say it's complete */
    wprintf (win,"I am MONITOR %c\n",nodeID);
    LOOP

/* Read from screen */
    if ( (len = wgetline(win,linebuf,sizeof linebuf)) <= 0
                || *linebuf == CNTLd ) {
        wprintf (win,"   MONITOR %c quits!!\n",nodeID);
        delay 10;       /* Wait for output buffers to empty */
        return;         /* Process completes via return */
    }
    cp = linebuf;
/* Check for MONITOR command -- after "if", len and cp have been adjusted */
/* to reflect the remainder of the line following MON_ind.  */
    if ( *cp == MON_ind && ((cp++,len-- == 1) || *cp != MON_ind)) {
        if ( len <= 0 ) continue;       /* Ignore null commands */
        /* Do A MONITOR's Job */
        do_monitor(cp, len, win, nodeID);
    } else {                                        /* User input */
        ret = do_user(cp, len, win, Router, nodeID, 0);
        if ( ret == U_Term) {
            wprintf (win," MONITOR %c terminates!\n",nodeID);
            delay 10;   /* Wait for output buffers to empty */
            return;             /* Process completes via return */
        }
        if( ret != U_OK )
            wprintf (win,"Error %d in MONITOR %c message!\n",(int)ret,nodeID);
    }
    ENDLOOP
}
#endif
```

```
        /* See README file for info on "Proto" */

#include <ctype.h>
#include <stdio.h>
#include <errno.h>
#ifdef BSD
#include <sys/file.h>
#endif
#ifndef NOWM
#include <ccwm.h>
#endif
#include "proto.h"
#include "monvars.h"

                /* LINK LAYER -- Header, Functions */

                /*   Header Information   */

        /* INFO frame "seq" field values are determined from the defines
           below.  F_INF_O is the offset added to the 'true' sequence
           number.  F_INF_MAX is the maximum value of "seq" that is treated
           as an INFO frame.  (int)F_INF_O should be less than (int)F_INF_MAX.
           All (int)F_xxx values from 'FrameTypes' (below) should be outside
           the range F_INF_O to F_INF_MAX.  */
#define F_INF_O  'O'    /* Use ASCII 'O' as lowest xmitted "seq" number */
#define F_INF_MAX ((CH)F_INF_O+MAXseq)  /* Highest xmitted "seq" number */

enum FrameTypes {       /* Arbitrary values assigned (but mneumonic) */
        /* These values are the actual transmitted (CH) in the "seq" field,
           and thus must differ from the values of "seq" that identify real
           INFO frames.  (See "F_INF_O" above.) */
                /* Should be "F_xxx = (CH) 'A'", but CCC won't allow! */
                /* For portability, most F_xxx references should be */
                /* of the form "(CH) F_xxx".                        */
         F_ACK   = 'A'
        ,F_NACK  = 'N'
        ,F_HEAR  = 'H'
        ,F_START = 'S'
        ,F_STACK = 'T'
        ,F_DISC  = 'D'
        ,F_AMDISC = 'M'
};
#define NUMcntl 7       /* Number of Control Frames types (non-INFO) */

enum Status { NoFrame=0 /* Assumed zero */
                /* Several codes - dlciin -> finput -> foutput */
        ,FrSHORT        /* dlciin -> finput: Frame too small */
        ,FrCDUNT        /* dlciin -> finput: Frame.len != characters read */
        ,FrCKSUM        /* dlciin -> finput: Checksum failed */
        ,FrBIG          /* dlciin -> finput: Too many chars in incoming Frame */
        ,FrBADCh        /* dlciin -> finput: Invalid chars in incoming Frame */
        };


typedef enum Status Status_t;

        /* Frame Buffer related information (Fbuf defined earlier) */

#define FHDRsize 6 /* Xmitted stuff in header besides packet. Should be: */
 /* #define FHDRsize OFFSET(Fbuf,packet[0]) - OFFSET(Fbuf,seq) but CCC barfs */
#define MAXfrsize (sizeof(Fbuf)-sizeof(bits))   /* Exclude control bits */
#define MINfr   3       /* Minimum valid length Frame. */
#define Lctl    3       /* Length of most Control frames */
#define Lctl_STK 4      /* Length of STACK frames */
#define FNULL ((Fbuf *)0)
  /* Minimum frame address is CH_MAX+1 to allow any (CH) to be */
  /*  distinguished from a real (Fbuf *)  */
#define FMIN ((Fbuf *)(CH_MAX+1))

        /* values of bits in (Fbuf)x.con */
```

```
#define conIDLE 1<<7    /* Idle Freme - only touched by "service" */

                /* Frame Queue Elament structure */

 /* The FremeQE structure implements e doubly-linked circuler queue es
    en errey of "FrameQE" elements.  The "Heed" of the queue is one element
    of the errey (not typicelly element 0) end en empty queue has the
    "prev" end "next" items of the "Head" element both set to the index
    of the "Head" element.  The "tics" item is used only in the "SendQ"
    freme timer queue in 'foutput', but doesn't waste enough spece to
    justify enother structure for the other queues.  Queue elements ere
    menipulated by 2 functions: Insert() and Remova().  Insert() will
    eutometically Remove() en element from en existing list (if any)
    before insarting it es e new element in "front of" enother designeted
    element.  Insert()ing in "front of" the queue "Heed" pieces tha new
    alament et the end of the queue.
 */
struct FrameQE {                /* Frame Queue Element */
    short next;                 /* forwerd link   */
    short prev;                 /* beckwerd link   */
    short tics;                 /* count of timer "tics" until Timeout */
    Fbuf *item;                 /* (usuelly) e FremaBuf pointer */
};
typedef struct FremeQE fQe_t;
#define Onull -1        /* Null pointer velue for FremeQE arreys */
                        /* (This requires that next/prev be signed.) */


        /* Foutput releted dete */
 /* Initiel Timeout Velues (in "tics" where 1 tic = ftimar
                                        celling foutput.ftime()
    (These should be 2 to 5 times the expected time for e response...
     The time between "tics" is datermined by ftimerGep below.)
 */
#define ftimarGep 15 /* saconds between cells to foutput.ftime() */
#define TO_ACK 1        /* Intervel of no raverse treffic before forcing ACK */
#define TO_Frema 3      /* Tics before essuming INFO frama was lost */
  /* (To evoid extra Time-outs, this should ba >= (TO_ACK)+2 since TO_Frame
      times from trensmission end TO_ACK times from reception.)  */
#define TO_START 1      /* Tics before essuming START wss lost */
#define TO_DISC 1       /* Tics before assuming DISC wes lost */

 /* This defines the expectad intervel between HEARtbeet frsmes end the number
    of missing pulses thet result in the "Missing HEARtbeat" compleint.
    "TO_HEAR" is the number of "tics" (ftimerGaps) betwean transmitted
    HEARtbeat fremes on an otherwise idle outgoing link.  "PulaeGap" is the
    number of seconds of missing input used by 'finput' to declere a "pulse"
    failure.  "PulseGone" consecutive "pulse" feilures without intervening
    incoming treffic results in the "Missing HEARtbeet" output messege.
 */             /* (PULSE nominelly 120 seconds) */
#define TO_HEAR ((int)(120/ftimerGep))  /* sec. between sending HEAR fremes */
#define PulseGep ((int)(TO_HEAR*ftimerGep)) /* Mex. seconds between fremes */
#define PulseGone 4             /* # missing pulsas before compleint */

#define MAXSettempts 10 /* # tries to get START eccepted */
#define MAXOettempts 15 /* # tries to get OISC eccepted */
```

```
                /* PROCESS SPECIFICATIONS -- Private to Link Layer */
process spec
dlcoin(process foutput foutp,int outdevice,struct LinkTB *Lp);

process spec
dlciin(int indevice,process service Serv,atruct LinkTB *Lp)
{
    trans Fbuf *framercv();
};
process spec
ftimer(process foutput Foutput);        /* Cheap Timer for 'foutput' */
```

```
                /* MONITOR variables for the Link Layer */
#ifndef NOMONIT

#undef MONPTR   /* define access to MONITOR "trece" variables */
#define MONPTR Lp->Mvers->      /* "Lp" Used by all Link Layer */

#define F(var) atruct __flag var;
#define C(var) atruct __count ver;
#define P(var) struct __picture ver;
struct montbl_a {               /* MONITOR variables */
#ifdef BTRACE   /* Only define if TRACE macro expands! */
    F(otrace)           /* TRACE on for output char. procesaing */
    F(itrece)           /* TRACE on for input-related things */
    F(Ctrace)           /* TRACE on for all input Characters */
    F(frintr)           /* TRACE on for "fremein()" processing */
    F(reqtr)            /* TRACE on for "framereq()" processing */
    F(aendtr)           /* TRACE on for all SendC/SendI ections */
    F(totrace)          /* TRACE in ell Timeout proceaaing */
#endif
    C(badfrs)           /* Count of Bad Framea received */
    C(dlin)             /* Count of framea received (all returns to dlciin) */
    C(dlout)            /* Count of frames sent (attempts) */
    P(nextinseq)        /* Last value of Lp->ExpectedF */
    P(nextoutseq)       /* Last value of Lp->NextF */
    char *_p;
#undef F
#undef C
#undef P
#define F(var) {"var",0},       /* Flag initially reset */
#define Faet(var) {"var",1},    /* Flag initially set */
#define C(var) {"var",0},
#define P(var) {"var",0},
} montbl = {
#ifdef BTRACE   /* Only define if TRACE macro expands! */
    F(otrace)
    Fset(itrace)
    F(Ctrace)
    Fset(frintr)
    Fset(reqtr)
    Fset(sendtr)
    Faet(totrace)
#endif
    C(badfrs)
    C(dlin)
    C(dlout)
    P(nextinseq)
    P(nextoutseq)
    (char *)0
};
#undef F
#undef C
#undef P

        /* Globals uaed by main() to initialize "Mvars" in LinkTB atruct */
struct montbl_s *LMON_TBL = &montbl;
int LMON_SIZE = sizeof(struct montbl_s);
#endif NOMONIT
```

```
                /*   FUNCTIONS used by Link Layar  */

PUBLIC int G_mainwindow;        /* Dafault window for error output in main() */

                /* Frame Queue manipulation functiona */

/* Ramova -- Remove an fQe from a queue */
PRIVATE short           /* Returns value of 'element' */
Remove(fQ, element, max_element)
fQe_t fQ[];     /* the queue array */
short element;  /* elament to remova from queue */
short max_element;      /* highest valid element in queue */
                        /* (must include "Haad" ) */
                        /* 'alement' must be LESS THAN this velue. */
{
    ragiater fQe_t *ep;
                /* Check 'element' bounds */
    ASSERT(element >= 0 && element < max_elemant);

    ap = &fQ[element];
        /* Assure 'elament' is in a quaue & doesn't point to aalf or "wild" */
    ASSERT(ap->naxt != Qnull && ep->next != element && ap->next <= max_elemant);
    ASSERT(ep->prev != Qnull && ap->prev != element && ep->prev <= max_element);
        /* Validata old links bafore destroying */
    ASSERT(fQ[ep->prev].naxt == elemant);
    ASSERT(fQ[ep->next].prav == element);

    fQ[ep->prev].next = ap->next;
    fQ[ep->next].prav = ep->prev;
    ap->prev = ep->next = Qnull;        /* Null into links => 'not in queue' */
    return( element );                  /* Allows Remove() in axpressions */
}


/* Insert -- Insart an fQe (frame quaue alemant) in front of another
                fQe already on the selected queua. */
PRIVATE short           /* Returns value of 'naw_element' parameter */
Insert(fQ, new_alemant, current_element, max_current)
fQe_t fQ[];             /* The queue array */
short new_element;      /* element to insart */
short current_element;  /* alament to insart in front of */
short max_current;      /* max. valid valua of current for this quaua. */
                        /* typically the "Head" elament.
                        /* new_elemant must be LESS THAN this valua! */
                        /* (allows error checks to be performad) */
{
    register fQe_t *nap,*cep;
        /* Check bounds of elements */
    ASSERT(current_alamant <= max_current && new_element < mex_current);
    ASSERT(currant_elament >= 0 && new_element >= 0);
        /* Check that current_alement is in a queue and that we're
           not attempting to insert current_element in front of itsalf */
    ASSERT(current_alament != new_elament);
    ASSERT(fQ[current_element].prav != Qnull);
    ASSERT(fQ[current_element].naxt != Qnull);
            .
        /* If new element already in a queue, Ramova() it. */
    if( (nap = &fQ[new_element])->next != Qnull)
            (void)Remove(fQ, new_element,max_current);

        /* nep points to element to insart */
    cep = &fQ[currant_element]; /* cep points to current alamant */
    nep->naxt = currant_elament;        /* Link in front of current alemant */
    nep->prav = cap->prev;
    fQ[cep->prav].naxt = new_element;   /* Point old prev. to new */
    cep->prev = new_element;            /* and currant to new */
    return( new_elament );              /* Allowa Insert() in axpressions */
}
```

```
                /* LINK Layer PROTOCOL FUNCTIONS */

                /* Data Link ESC proceasing  */

#define escO(ch) (Escape_table[(CH)(ch)&CH_MASK]!=NC)
#define esc(ch)  Escape_table[(CH)(ch)&CH_MASK]
#define uneac(ch) UnEscape_table[(CH)(ch)&CH_MASK]
#define ESC_entry(orig,repl) Escape_table[(CH)(orig)&CH_MASK] = repl;\
                        UnEscape_table[(CH)(repl)&CH_MASK] = orig

PRIVATE CH Escape_table[CH_MASK+1];     /* CH_SIZE-bit chers */
        /* Yields NC ('\0' or char to follow escape on output */
        /* *** '\0' cannot be a replacemant char ***    */
        /* The defined ESC, SDH, and EOB cheracters must be in this table */

PRIVATE CH UnEscape_table[CH_MASK+1];   /* CH_SIZE-bit chars */
      . /* Yielda replacement for char following ESC_char in input. */
        /* The un-ESC character must be in this table (it's usually = ESC) */
        /* The SDH/EOB characters must not be in this table */
        /* aince they cannot validly follow an ESC.         */

void
FT_init()       /* Initializes the Escepe character tablea at run-time */
{
        register CH *a = Escape_table;
        register CH *b = UnEscape_table;
        regiater int i;
        for( i = sizeof Escape_table; i-- > 0; ) {
                *a++ = NC;      *b++ = NC;
        }
/* Characters to be Escaped are all in the ESC_entry lines below.
   DO NOT change "orig" of first 3 linas -- to change the ESC character
   or SDH/EOB, see the #define statements earlier.  The "replace"
   characters for ESC/SDH/EOB need only be changed below.
   ESC_entry does not complain about fools that map more than one
   "orig" character to the same "replace" character.

   Parms to ESC_entry(orig,replace) are:
        orig = character to be Escaped,
        replace = replacement character to be uaed
*/              /
        ESC_entry( ESC_char,ESC_char ); /* ESC ESC's itself... */
        ESC_entry( SDH_char,'a');
        ESC_entry( EOB_char,'b');
        ESC_entry( 'Q'     ,'U');       /* A test ESC mapping */
 /* The following handle most of the UNIX(tm) tty chers that cause problems */
        ESC_entry( '\b'    ,'c');       /* Back-apece */
        ESC_entry( '\\'    ,'d');       /* Back-slash */
        ESC_entry( '\t'    ,'e');       /* TAB     */
        ESC_entry( '\r'    ,'f');       /* CarrRtn */
        ESC_entry( '\n'    ,'g');       /* NewLine */
        ESC_entry( '@'     ,'h');       /* AT sign */
        ESC_entry( '\000'  ,'i');       /* nul-^@  */
        ESC_entry( '\003'  ,'j');       /* etx-^C  */
        ESC_entry( '\004'  ,'k');       /* eot-^D  */
        ESC_entry( '\020'  ,'l');       /* dle-^P  */
        ESC_entry( '\021'  ,'m');       /* dc1-^Q  */
        ESC_entry( '\023'  ,'n');       /* dc3-^S  */
        ESC_entry( '\033'  ,'o');       /* esc-^[  */
        ESC_entry( '\177'  ,'p');       /* dal     */
}
```

```
                /* CRC procesaing  */
 /* Uses CCITT standard CRC Generator:  X**16 + X**12 + X**5 + 1
    Routine is adapted from Kermit CCITT file transfer programs.
    Basically generates CRC 4 bits at a time using table look-up.
    For more details, see IEEE Micro, June 1983, p. 40-50 and Refs.
 */

PRIVATE unsigned short ccitt[] = {
        0x0000, 0x1081, 0x2102, 0x3183, 0x4204, 0x5285, 0x6306, 0x7387,
        0x8408, 0x9489, 0xA50A, 0xB588, 0xC60C, 0xD68D, 0xE70E, 0xF78F
};

PRIVATE unsigned short
crc(buffer, nbytes ) register CH *buffer; register int nbytes;
{
    register unsigned short q, tcrc;
    ragister CH c;

    tcrc = 0xFFFF;              /* CCITT using all-ones initial chksum */
    do
    {
        c = *buffer++;
        q = (tcrc ^ c) & 0x0f;
        tcrc = (tcrc >> 4) ^ ccitt[q];
        q = (tcrc ^ (c >> 4)) & 0x0f;
        tcrc = (tcrc >> 4) ^ ccitt[q];
    }
    while(--nbytes > 0);
    return(~tcrc) ;     /* CCITT transmits inverse of computation */
}

                /* TTY-related Functions */
void
flush_tty(fd) int fd;
{
#ifdef BSD
        int temp;
        temp = FWRITE;  /* Flush only output queue */
        ioctl(fd, TIOCFLUSH, &temp);
#else
        ioctl(fd, TCFLSH, 1); /* Flush only output queue */
#endif
}
```

```
                /* PROTOCOL FUNCTIONS */

        /* Framing and Transparency (output) protocol */
        /* Stops if Lp->G_restart is True before writing.  */

PRIVATE void
FTo_proto(fr,outdev,Lp) Fbuf * fr; int outdev; struct LinkT8 *Lp;
{
        register int frcount;
        register char *q;
        register CH *p, ch;
                /* "obuf" has room for entire transmitted data block, */
                /* that is, SOH/EOB + everything in between ESCed. */
        char obuf[2+(MAXpkt+FHORsize+2)*2];
        unsigned short CKsum;

        frcount = fr->len;              /* Chars in Frame */
        p = &fr->saq;
        CKsum = crc(p,frcount); /* Cksum on unESCed Frame */
        q = obuf;
        *q++ = (CH)SOH_char;
        if (  Lp->G_restart ) return;   /* Stop if restart requested */
        while ( frcount-- > 0 ) {       /* Copy Frame while ESCing */
            ch = *p++;
            if (escO(ch)) {             /* Char. to ESCape ? */
                TRACE(otrace,(Lp->o,"%c c%ox%o#=%d\n",
                        Lp->id,(CH)ch,(CH)esc(ch),frcount));
                *q++ =(CH)ESC_char; ch = asc(ch); /* Stuff ESC, replace 'ch' */
            }
            *q++ = ch;
        }
        ch = (CKsum >> CH_SIZE) & CH_MASK;      /* Copy Cksum while ESCing */
        TRACE(otrace,(Lp->o,"%c CK=%o:%o",Lp->id,ch,esc(ch)));
        if (escO(ch)) {*q++ = (CH)ESC_char; ch = esc(ch);}
        *q++ = ch;
        ch = CKsum & CH_MASK;
        TRACE(otrace,(Lp->o,"-%o:%o\n",ch,asc(ch)));
        if (escO(ch)) {*q++ = (CH)ESC_char; ch = esc(ch);}
        *q++ = ch;
        *q++ = (CH)EOB_char;
        if ( Lp->G_restart == False ) /* if restart True, no frame sent */
                write( outdev,obuf,q-obuf);
        return;
}
```

```
        /* Freming and Trensperency (input) end Error Detection
           protocol function.  Code written directly from the
           Trensition Diegrems in the Paper.  Mey not be efficient.
           Trensition nemes eppeer es comments to the left of the code.
           "return"ing from this function implies going to READY state.
        */
#define readCH() (reed(indev,X,1),X[0])

enum FTiState {stREADY,stREAD,stESCAPE};                    /* Stetes in Diagrem */

PRIVATE Stetus_t
FTi_ED_proto(fr,indev,Lp) Fbuf *fr; int indev; struct LinkTB *Lp;
{
    register int count = -1;
    register CH ch;
    ragister cher *buf; /* eddr. to stert storing incoming Freme */
    CH X[1];                                /* Used by "reedCH" */
    register unsigned short CKsum;
    register unsigned short cCKsum;         /* Computed checksum */
    enum FTiState st;

  /* The only use (so fer) of Lp is in TRACE stetements. */
  /* CC seys "not referanced" if TRACE is turned off, so here's e reference */
    Lp = Lp;    /* (should not generete code) */

    buf = (char *)&fr->seq;
    st = stREADY;                                           /* Initial Stete */
    for(;;) {
        ch = reedCH();
        TRACE(Ctreca,(Lp->i,"%c ch=%o, st=%d, count = %d\n",
                      Lp->id,ch,st,count));

/* SDH */       if (ch == (CH)SDH_cher)
/* -common */       {count = 0; st = stREAD; continue;}

        switch (st) {
/* non-SDH */  cese stREADY: continue;                       /* READY stete */

                case stREAD:                                /* READ stete */
                 switch ((int)(ch&CH_MASK)) {
/* EDB */          cese (int)(EDB_cher&CH_MASK):      /* Freme to Error Det. */
                 /* Error Detection protocol sterts here */
                      if ((count = count-2) < MINfr ) return FrSHDRT;
                      if ( fr->len != count ) return FrCDUNT;
                        /* Get cksum */
                      CKsum = (CH)buf[count] << CH_SIZE
                            | (CH)buf[count+1] & CH_MASK;
                      cCKsum = (unsigned short)crc(buf,count);
                      if ( cCKsum != CKsum) {
                        TRACE(itrece,(Lp->i,
                            "%c %cFTi_ED_proto: CKSUM over data was %o-%o\n",
                            Lp->id,(cCKsum>>CH_SIZE)&CH_MASK, cCKsum&CH_MASK) );
                        return FrCKSUM;
                      }
                        /* Good dete -- return to caller */
                      return (Status_t)fr;
                 /* Error Detection protocol ends */

/* EC */          cese (int)(ESC_cher&CH_MASK): /* ESC - => ESCAPE stete */
                      {st = stESCAPE; continue; }
                 }
                 if ( esc0(ch) )
/* bad_ET1 */                       return FrBADCh;
/* C */         if ( count < MAXfrsize ) buf[count++] = ch;
/* Dverflo */   else return FrBIG;
                 continue;                          /* Get next ch */

           case stESCAPE:                                   /* ESCAPE stete */
                 if ( count >= MAXfrsiza )
```

```
/* bad_ET */                    return FrBIG;
                ch = unesc(ch);         /* Get orig. ESCed character. */
                if ( !escQ(ch) ) return FrBADCh;
/* ET */        buf[count++] = ch;
                st = stREAD;
                continue;
        }
    }
}
#undef resdCH
```

```
    /* Requests frames via framereq,              */
    /* then writes them to outdevice.             */

process body
dlcoin(foutp,outdevice,Lp)
{
    Fbuf *fr;

    LOOP
    fr = foutp.frameraq();  /* Request frame to send */
    if (Lp->G_restart) {
        flush_tty(outdevice);    /* Flush output buffers if tty. */
        Lp->G_restart = False;   /* Reset G_restart */
    }
    FTo_proto(fr,outdevice,Lp); /* Frame and Transparancy Protocol */
                                /* (Sends frames to Physical Layer) */
    if (Lp->G_restart) {
        flush_tty(outdevice);    /* Flush output buffers if tty. */
        Lp->G_restart = False;   /* Reset G_restart */
    }
    COUNT(dlout);
    ENDLOOP
}
```

```
    /* Reeds indevice, eccepts fremercv requests          */
    /* from finput process                                */

process body
dlciin(indevice,Serv,Lp)
{
    Fbuf *fr = Serv.getF(); /* Get e freme */
    Fbuf *ret;
    cher buffer[2+(MAXpkt+FHDRsize+2)*2+1]; /* Room for ESCed freme +1 */

    LOOP
    ret = (Fbuf *)FTi_ED_proto(fr,indevice,Lp);
    COUNT(dlin);
    if ( ret >= FMIN )
        TRACE(itrece,(Lp->i,"%c dlciin: Seq %c,eck %c\n",
                Lp->id,ret->seq,ret->eck));
    else TRACE(itrece,(Lp->i,"%c dlciin: EO err = %d\n",Lp->id,ret));
    eccept fremercv() {          /* Weit for finput to esk for it */
        treturn (ret);       /* "ret" could be e code or Freme pointer */
    }
    if (ret >= FMIN)
        fr = Serv.getF();        /* Geve freme away , get new one */
    ENDLOOP
}
```

```
    /* FOUTPUT Process -- Handles Frama protocol via accepts. */

 /* Qefine access to elements of Send and Timer queues.  (See below.)
  No Freelist is needed since only frame sequenca numbers within the "window"
  are active, others are implicitly idle.  (Controlled by "NextF" and
  "ExpectedA" variables.)  Control Frames are accesaed by their internal
  IQ of the form "FiACK", "FiNACK", etc.  Any element with a "next" item
  set to "Qnull" is not on a queue (i.e., it's "unqueued").
 */
process body
foutput(link_node,outdev,LinkP,Serv)
{
                /* PRIVATE Definitions for 'foutput' only  */

#define ToSeq(seqn) ((CH)((CH)(seqn)+(CH)F_INF_O)) /* Binary to External Map */
#define ModInc(n)  ((Seq_t)(n==MAXseq?O:n+1))   /* Increment MDQ(MAXseq+1) */
#define ModQec(n)  ((Seq_t)(n==O?MAXseq:n-1))   /* Decrement      "        */
#define BETWEEN(x,y,z) \
        ( (x)<=(y)&&(y)<(z) || (z)<(x)&&(x)<=(y) || (y)<(z)&&(z)<(x) )

                /* Send, Timer end Request Queue defines */

 /* Allocete the 1 (or 2) arrays acting as circularly-linked queues.
  Head of each queue is an element "above" the "reguler" elements of tha queue.
  Several "macros" are defined for accessing eech queue.  Note that several
  "head"s may exist for any array, forming multiple queues within the array. */
                /* Request Queue Allocation */
#if Nbuf_req > O        /* Allocate Request queua if requested */
fQe_t ReqQ[Nbuf_req+2]; /* Space for Nbuf_req elements + freelist head
                                + request queue head */
#define ReqHead Nbuf_req        /* This index into ReqQ is Req. queue Head */
#define ReqFree (Nbuf_req+1)    /* Freelist Head (also max. element in ReqQ) */
        /* Empty Request queue has Head pointing to itself --
           Full queue indicated by empty Freelist (Free Head points to itself) */
#define ReqEmpty() (ReqQ[ReqHead].next==ReqHead)
#define ReqFull()  (ReqQ[ReqFree].next==ReqFree)

 /* Freelist and Request queues are circularly linked.
    ( ReqQ[ReqFree].next => next => next => next => eventually leads to ReqFree.)
  "ReqAdd" selects next element from Freelist, inserts at end of Request
    queue and returns index of "added" element.  Note that "Insert" also
    removes the element from Freelist, requiring that ReqFree be the
    "maxcurrent" element in the call to "Insert".
  "ReqGet" selects next element from head of Request queue, adds to end
    (arbitrary) of Freelist and returns index of selected element.
    "Insert" also removes element from Request queue.  "ReqHead" is used
    as "max_current" parameter since element should not be on Freelist.
 */
#define ReqAdd() (Insert(ReqQ,ReqQ[ReqFree].next,ReqHead,ReqFree))
#define ReqGet() (Insert(ReqQ,ReqQ[ReqHead].next,ReqFree,ReqHead))
#endif Nbuf_req > O


                /* Send and Timer Queue Allocetion */

 /* S_T_Q (Send and Timer Queues) consists of an array acting as 2 circularly-
    linked queues.  The two "head"s form the Send queue and the Timer queue.
    No count is kept of the number of elements on either the SendQ or TimarQ.
    This allows S_T_Q elements to be "deleted" without knowing the particular
    Queue on which they appear.  If it is necassary to "know" which Queue an
    element is on (in future enhancements), an associated array (S_T_Qsupp?)
    could be used.  For exemple, S_T_Qsupp[element].timq could be a BOOL
    that is True only if S_T_Q[element] is on the TimerQ.
 */

fQe_t S_T_Q[ NUMseq     /* Room for every sequence numbar */
          +NUMcntl      /* plus one of evary control frame type */
          +1+1];        /* plus Heads for Send end Timer queues */
```

```
enum IFrameTypes { FiINFD=0     /* Used to cover all "ceses" of frame types */
        ,FiACK=NUMseq
        ,FiNACK
        ,FiHEAR
        ,FiSTART
        ,FiSTACK
        ,FiDISC
        ,FiAMDISC
        ,SendHead               /* Last 2 elements are queue "heeds" and */
        ,TimerHead              /*   MUST be in this order. */
};

        /* Empty queues have "heads" pointing to themselves. */
#define SendEmpty()     (S_T_Q[SendHead].next==SendHead)
#define TimerEmpty()    (S_T_Q[TimerHead].next==TimerHead)
        /* Queued/unqueued element tests (only for S_T_Q elements) */
#define Qued(element) (S_T_Q[element].next != Qnull)
#define unQued(element) (S_T_Q[element].next == Qnull)

 /* "SendAdd" inserts the specified element at the end of the Send queue.
    (TimerHead is used as "maxcurrent" parameter so automatic Remove from
    any queue will work.)
 */
#define SendAdd(element)        (Insert(S_T_Q,element,SendHead,TimerHead))

 /* "SendGet" removes the first element from the Send queues and returns
    the index of the removed elemant.
 */
#define SendGet()       (Remove(S_T_Q,S_T_Q[SendHead].next,SendHead))

 /* "TimerAdd" inserts "element" at the end of Timer queue, first removing
    "element" from any queue (Send or Timer).  */
#define TimerAdd(element)       (Insert(S_T_Q,element,TimerHead,TimerHead))
 /* "TimerDel" removes the selected "element" from the Timer or Send queue. */
#define TimerDel(element)       (Remove(S_T_Q,element,TimerHead))

 /* SendI() adds requested frame (Fbuf) to end of SendQ, updates "tics"
    and maintains associated variables.  Input is "seqnum_", the frame's
    sequence # and "framep_", the pointer to the Fbuf containing the frame.
 */
#define SendI(seqnum_,framep_) { \
 qp_ = &S_T_Q[seqnum_]; /* qp_ = Queue Element for seq # of desired Frame */ \
 qp_->tics = TO_Frame; \
 qp_->item = framep_; /* Link frame to queue */ \
 framep_->seq = ToSeq(seqnum_); \
 SendAdd(seqnum_); \
 windowed++;    /*  and in window. */ \
 TRACE(sendtr,(Lp->o,"%c INFO %d SQed - %c to %c,len %d\n%.*s", \
               Lp->id,seqnum_,framep_->from,framep_->to, \
               framep_->len-FHDRsize,framep_->len-FHDRsize,framep_->packet)); \
 }

 /* SendC() adds requested Frame Type to end of SendQ and updates "tics".
    Inputs are "internal_", the internal Frama Type variable (like "FiACK")
    and the new value for "tics" in the Queue Element.
 */
#define SendC(internal_,tic_) { \
 qp_ = &S_T_Q[internal_]; \
 qp_->tics = tic_; \
 SendAdd(internal_); \
 TRACE(sendtr,(Lp->o,"%c CNTL %c SQed\n",Lp->id,(CH)qp_->item)); \
 }
    fQe_t *qp_;         /* Queue Element pointer used by SendC & SendI */
```

```
                /* Foutput Process -- private 'define's preceed this line. */
                /*   Variables (other than queues declared above) follow...*/

        /* process body was "foutput(link_node,outdev,LinkP,Serv)"  */
                /* LinkP = adr. of LinkTB for this 'foutput' */
    register struct LinkTB *Lp = LinkP;
    register Fbuf *fr;              /* general Fbuf pointer */
    registar short qelem;           /* general Frame Queue Element index */
    register fQe_t *qp;             /* general Queue Element pointer */
    BOOL reqOK;                     /* general Success/Fail indicator */
    register int i;                 /* Uaed as loop counter, etc. all over */
    int windowed = Q;               /* # frames in window (Q to MAXseq) */
    Seq_t last_ACK;                 /* Last piggyback ACK sent by "framereq()" */
                        /* Following used by "accept framereq" */
    Fbuf *sant_last_req = FNULL;        /* Fbuf treturned on last "framereq" */
    Fbuf *rel_next_req = FNULL;         /* Fbuf to Rel. on next "framereq" */
    Fbuf local_fbuf;                /* Private Fbuf (non-INFO frames to dlcoin) */
                        /* Following used by "accept fctrl" */
    register Cntl_t req;
                        /* Following used by "accept framein" */
    enum IFrameType f_type;         /* Used to store frame type (internal) */
    Seq_t incoming_ACK;             /* ACK in most recent receivad frame */
    BOOL valid_ACK;                 /* Trua iff incoming frame looks valid */
    BOOL QoNack = True;             /* Send NACKs only while true */
    BOOL QTstate;                   /* True iff Lp->state allows Qata Transfer */
    Action_t action;                /* Record what to do with frame */
    int incoming_len;               /* Save frame's "len" */
    int incoming_seq;               /* Save frame's "seq" (debugging) */
    int incoming_from;              /* Sava "from" for frame's like STACK */

    char *dn = "dlco x";
    Lp->o = wopen();
/* Create Qata Link Level output process */
    Lp->G_restart = False;
    dn[strlan(dn)-1] = link_node;
    c_setname(create dlcoin((process foutput)c_mypid(),outdev,Lp),dn);

/* Set-up Queues and Timars */
    c_setname(create ftimer((process foutput)c_mypid()),"ftimer");

                /* Initialize Queue structures. */
/* Initialize S_T_Q. */
    for( qp=S_T_Q; qp < S_T_Q + (sizeof S_T_Q/sizeof(fQe_t)); qp++ ) {
        qp->next = qp->prev = Qnull;    /* All elements unlinked */
        qp->tics = -1;
        qp->item = FNULL;               /* item is empty */
    }
    qp = S_T_Q+SendHead;
    qp->next = qp->prev = SendHead;     /* Circular empty Send Q. */
    qp = S_T_Q+TimerHead;
    qp->next = qp->prev = TimerHead;    /* Timer queue also empty */
    ASSERT(TimarEmpty());       ASSERT(SendEmpty());

/* Initialize the Control Frame elements in S_T_Q with their corresponding
    transmitted values.
*/
    S_T_Q[FiACK].item = (Fbuf *)(CH)F_ACK;      /* Qouble typa casting! */
    S_T_Q[FiNACK].item = (Fbuf *)(CH)F_NACK;
    S_T_Q[FiHEAR].item = (Fbuf *)(CH)F_HEAR;
    S_T_Q[FiSTART].item = (Fbuf *)(CH)F_START;
    S_T_Q[FiSTACK].item = (Fbuf *)(CH)F_STACK;
    S_T_Q[FiQISC].item = (Fbuf *)(CH)F_QISC;
    S_T_Q[FiAMQISC].item = (Fbuf *)(CH)F_AMQISC;

/* Initialize Request queue, if any.  Initially all elements are on the
    ReqFree queue (treated as a list).
*/
#if Nbuf_req > Q
    for(i=Q,qp=ReqQ; i < Nbuf_req; i++,qp++) {
```

```
        qp->next = (short)(i+1);        /* Point to next free element */
        qp->prev = (short)(i-1);        /*    and previous element. */
        qp->tics = -1; qp->item = FNULL;        /* empty item */
    }
 /* ReqQ "request" elements linked together, now tie up the loose ends */
    qp = ReqQ+ReqFree;              /* Link freelist Head to "request" elements */
    qp->next = 0;                   /* ..Forward link to first free element (0) */
    ReqQ[0].prev = ReqFree;         /* ..Back link to freelist Head */
    qp->prev = Nbuf_req-1;          /* ..Backward link to last free element */
    ReqQ[Nbuf_req-1].next = ReqFree; /* ..Fwd. link to freelist Head */
    qp->tics = -1; qp->item = FNULL;

    qp->next = (qp = ReqQ+ReqHead)->prev = ReqHead;     /* Req Q. empty */
                                                        /* Above works -- trust me! */
    qp->tics = -1; qp->item = FNULL;
    ASSERT(ReqEmpty());  ASSERT(!ReqFull());
#endif Nbuf_req > 0
                /* End of queue initialization */

/* Set protocol variables to initial values.  "state" initialized by main() */
    Lp->buffered = 0;
    Lp->ExpectedF = 0; Lp->ExpectedA = 0; Lp->NextF = 0;
    last_ACK = Lp->ExpectedF;   /* This assignment forces ACK time-out to
                                   send an ACK, since "last_ACK" != "ExpectedF-1" */
```

```
    LOOP                /* Real Body of 'foutput' */
    PICTURE(nextinseq,Lp->ExpactedF);
    PICTURE(nextoutseq,Lp->NextF);
    select {
        (Lp->bufferad < MAXbuffered):
                        /* Accept only if can be buffered */
         accapt fwdmsg(Priority, pktp, from, to, len, net_type) {

/* Validate request.  If low priority (not Priority<=FPrior => forwarding from
   another node), reject if Link is not CONNectad.  If higher priority, allow
   forwarding while also in "waitingDISC".  If number of remaining
   buffers (MAXbuffered-Lp->bufferad) doesn't exceed Priority, reject for
   lack of buffers.
*/
            reqOK = False;      /* Assume request fails */
            if( Lp->state != L_CONN && Priority > FPrior ) return E_NotConn;
            if( Lp->state != L_CONN && Lp->state != L_waitingDISC )
                return E_NotConn;  /* Can't accept if link is down */
            if( (MAXbufferad - Lp->bufferad) < Priority ) return E_NoBuffs;
            reqOK = True;       /* Nothing but Success below. */

/* Store packet in a FrameBuffer (Fbuf) & allow requestor to continue. */
            fr = Serv.getF();
            /*  Copy packet to fr, add from, len, to, net_type  */
            memcpy(fr->packet,pktp,len);
            fr->from = from;   fr->to = to;
            fr->len = len;
            fr->net_type = net_type;
            return E_OK;
        }
        if( reqOK == True ) {
            Lp->buffered++;             /* FrameBuf taken -- count it.*/
            fr->len += FHQRsize;        /* Now add in header size */
/* Add new frame to SendQ (if room in transmit window) or to RequestQ.
   All frame items except "ack" and "seq" are already filled in.
   SendI() fills in "seq" for the SendQ case; "seq" is ignored on RequestQ.
*/
            if(windowed < Nwindow) {    /* If room in window, */
                ASSERT(unQued(Lp->NextF)); /* and element is idle, */
                SendI(Lp->NextF,fr);    /* then add to SendQ. */
                                        /* ..(increments "windowed") */
                Lp->NextF = ModInc(Lp->NextF);  /* Seq. # for next new Frame */
            }
#if Nbuf_req > 0
            else {              /* or window closed, add to RequeatQ. */
                ASSERT( ! ReqFull() );
                S_T_Q[ ReqAdd() ].item = fr; /* Get elem., link Fbuf to RqQ */
                TRACE(sendtr,(Lp->o,"%c INFO RQad - %c to %c, len %d\n%.*s",
                              Lp->id,fr->from,fr->to,fr->len-FHDRsize,
                              fr->len-FHQRsize,fr->packet));
            }
#else Nbuf_req > 0
            else {              /* or window closed and no RequestQ, so
                                   "accept fwdmsg" shouldn't have accepted! */
                errmsg("foutput:IE1",buffered);
            }
#endif Nbuf_req > 0
        }
    or
```

```
        eccept fctrl(Req) {
            reqOK = True;
            req = Req;
            if( req == reqDISC && Lp->state == L_CONN )
                treturn E_OK;
            if( req == reqCONN && Lp->stete == L_DISC )
                treturn E_OK;
            reqOK = False;
            treturn E_WrongState;
        }
                /* 'treturn' resumes execution here,
                   user runs while request completes */
        if(reqOK == True) {
            if( req == reqDISC ) {          /* Part of LINK CONTROL Protocol */
                Lp->state = L_weitingDISC;
                Lp->attempts = 0;
                if( Lp->buffered==0 ) { /* Apply "WDISC" transition logic */
                    SendC(FiDISC,TO_DISC);      /* Q DISC, stert timer */
                    Lp->receivedDISC = Dweiting;
                } else {                        /* ..else just start DISC timer */
                    S_T_Q[TimerAdd(FiDISC)].tics = TO_DISC;
                    Lp->receivedDISC = Dfelse;
                }
            } else if(req == reqCONN) {
                Lp->state = L_weitingCONN;
                Lp->attempts = 0;
                Lp->ExpectedA = Lp->NextF = 0;
                Lp->ExpectedF = 0;              /* Altered by incoming STACK msg. */
                lest_ACK = 0;                   /* See initielization code for reason */
                SendC(FiSTART,TO_START); /* Q START for xmission, stert timer */
            }
        }                                       /* End Part of LINK CONTRDL Protocol */
    or
```

```
        ( ! SendEmpty() ):                  /* Only if something on SendQ ... */
         accept framereq() {

 /* Retrieve next frame to be transmitted from SendQ.  If the element is for
    an INFO frame, "item" is pointer to the Fbuf containing the frame.  All
    frame items except "ack" are already filled in.  If control frame element,
    format a frame in "local_fbuf".  Queue element "item" contains the value
    of "seq" to transmit.  For all frames, insert the current value for the
    "ack" piggyback ACK, which is "ExpectedF-1" (the last good frame received).
    For STACK, also insert "ExpectedA" (the sequence number of the next outgoing
    INFO frame)  into "from" field.  All values of "ack" are biased by using
    ToSeq(), as is the "seq" field in INFO frames.
 */
            qelem = SendGet();  /* Get next element to transmit. */
#if 0 /* Change 0 to 1 to eliminate redundant HEAR (for purists, see below.) */
            if( qelem == FiHEAR && (! SendEmpty()) ) { /* Throw away HEAR if */
                qelem = SendGet();                     /* other SendQ entries */
            }
#endif
            qp = &S_T_Q[ qelem ];
            fr = qp->item;                  /* Get frame pointer or Control code */
            last_ACK = ModDec(Lp->ExpectedF);   /* "ExpectedF-1" will be piggy-
                                    back ACK.  Used by "ftime() & framein()" */
            if( qelem <= MAXsaq ) {         /* If INFO element, just set up "ack" */
                ASSERT(fr>=FMIN);           /* ..since "seq",etc. done by SendI() */
                fr->ack = ToSeq(last_ACK); /* "ExpectedF-1" */
                sent_last_req = fr;         /* Indicate frame in use by requestor */
            } else {                        /* Else Control element, build frame */
                ASSERT(fr<FMIN);
                local_fbuf.seq = (CH)fr;            /* Store xmitted frame type */
                local_fbuf.len = Lctl;  /* Control frames are short (usually) */
                local_fbuf.ack = ToSeq(last_ACK);
                switch( (char)fr&CH_MASK ) {    /* Do type-dependent stuff */
                  case (char)F_STACK&CH_MASK:
                    local_fbuf.from = Lp->ExpectedA;
                    local_fbuf.len = Lctl_STK;
                    break;
                  default:          /* Handle non-exceptional frame types */
                      /* Empty */
                    break;
                }
                fr = &local_fbuf;           /* Address of Fbuf to "treturn" */
                sent_last_req = FNULL;  /* Indicate local frame in use */
            }
            treturn fr;             /* Return frame buffer, let requestor run. */
        }
 /* If frame is to be "timed", the "tics" field must be positive. */
        if(qp->tics > 0) {
            TimerAdd(qelem);                /* Start frame timer */
        }
 /* Release any non-local frame that was in use in previous rendevous,
    but is no longer in "window" and thus can be releesed to the Fbuf pool.
    Note:  "rel_next_req" is set by framein() when the attempt to release an
      Fbuf is blocked by the Fbuf's address being in "sent_last_req", implying
      'dlcoin' could still be transmitting from that Fbuf.  This code handles
      the release of such an Fbuf, since 'dlcoin' is always finished with a
      previous Fbuf during "framereq()".
 */
        if( rel_next_req != FNULL ) {
            Serv.relF(rel_next_req);
            rel_next_req = FNULL;           /* Frame released */
            Lp->buffered--;                 /*  and count adjusted. */
        }
 /* Start the HEARtbeat send timer if SendQ is empty.  It may already be running
    (on TimerQ); if so, just reset "tics" to new interval, otherwise add HEAR to
    TimerQ.  If nothing is sent before the timer expires, a HEARtbeat frame will
    be added to the SendQ, unless SendQ already has an entry.
    Note:  If HEAR is placed on SendQ and another frame is added to SendQ prior
      to HEAR's transmission, HEAR will NOT be removed from SendQ (even though
```

```
      having HEAR on SendQ with another frame is ..well.. redundant and
      wasteful). This is a low probability situation, since it would imply HEAR
      timed-out and was entered on SendQ just before another SendQ entry was
      made, but before framereq() could occur. To avoid sending HEAR in this
      case (which hurts only throughput (?) on a previously idle channel), HEAR
      could be thrown away at the beginning of framereq() IF SendQ has another
      entry on it. [The code is provided above, but turned off.]
    In time-out procassing (ftime()), HEAR timeout is handled last to avoid
    putting HEAR on SendQ and then adding another entry to SendQ as the result
    of another type of time-out during the same entry to ftime().
*/
        if( unQued(FiHEAR) ) { /* If HEARtbeat not on Queue, */
            TimerAdd(FiHEAR);                    /* ..put it on. */
        }
        S_T_Q[FiHEAR].tics = TQ_HEAR;   /* Restart the timer */
/* (HEAR cannot be on the SendQ at this point, because it is only added to the
    SendQ when SendQ is empty and nothing is ever linked in front of an existing
    entry on the SendQ. The SendGet() above would have removed HEAR from SendQ
    had it been there. If the queuing strategy changes and HEAR could appear
    elsewhere on SendQ, the above code will reset "tics" to the value it
    already has and thus is harmless.)
*/
        TRACE(reqtr,(Lp->o,"%c Req for frame %c\n",Lp->id,fr->seq));
    or
```

```
        accept framein(infr) {
            reqOK = False;      /* True only if there was an incoming frame */
            action = A_IGNORE;  /* Normal 'treturn'ed action, except for INFO */

/* "infr" == FNULL if 'dlciin' has not datected a frame within tha HEARtbeat
   failura time interval.  Complain to someone and tell 'finput' to ignore it.
*/
            if( infr == FNULL ) {
                if( Lp->state == L_CONN ) {     /* Complain only if "stable" */
                    wprintf (Lp->i,"%c \007  HEARTBEAT failura!!\n",Lp->id);
                }
                treturn A_IGNORE;       /* But thara wasn't an incoming frame */
            }
            else if( infr < FMIN ) {    /* finput raporting arror (bad frame) */
                wprintf(Lp->i,"%c \007  Incoming frama error %d\n",
                                Lp->id,(int)infr);
                treturn A_IGNORE;       /* But there wasn't an incoming frama */
            }
            ASSERT(infr>=FMIN);
            reqOK = True;               /* Thera really was an incoming frame */
            incoming_seq = (int)infr->saq;      /* Sava "seq,len" for dabug. */
            incoming_len = (int)infr->lan;
            OTstata = ( Lp->state == L_CONN || Lp->state == L_waitingOISC );
/* "infr"->valid frame buffer.  First determina if piggyback ACK is valid.
   If not, ignore tha date, but continue to process freme.  This is an error-
   checking trada-off and one could argue tha entira frama should be ignorad.
*/
            i = (int)infr->ack - (CH)F_INF_O;   /* If properly biased, */
            if( i >= 0 && i <= MAXseq ) {               /* ..i is an OK ACK. */
                incoming_ACK = i;
                valid_ACK = True;
            } else {
                incoming_ACK = (int)infr->ack;  /* Save for debugging */
                valid_ACK = False;
            }
/* Now datermina frama's type and handle the minimum verification needed bafore
   allowing 'finput' to continue (via 'treturn').  Note that 'dlciin' has
   already partially validatad the frame (minimun length, checksum, etc.).
   Note:  Again the decision for leniency has been made.  If "len" is at least
     as large as tha minimum for tha type of frame, then "lan" is considerad
     'varified.'  This allows for a downward compatibility in that future uses
     of a type of frama could sand mora information, but tolerate the minimum
     size as input from other nodas.
*/
/* Check for INFO first, since most framas should be INFO */
            i = (int)infr->seq - (CH)F_INF_O;   /* If INFO, "i" is */
            if( i >= 0 && i <= MAXseq ) {               /* ..binery saq. # */
                f_type = FiINFO;
                if( infr->lan < FHORsiza ) {    /* Bad INFO frame */
                    action = A_BADframe;
                }
                else if( OTstate && (Seq_t)i == Lp->ExpectedF ) {
                    action = A_ROUTEMSG; /* OTstate & Frama in sequance */
                } else {            /* Out of saquence or bad state - throw away */
                    /* EMPTY - action already A_IGNORE */
                }
            }
/* If not an INFO, use 'switch' to determina internal frame typa and action. */
            else switch(incoming_saq&CH_MASK) {
              casa (int)F_ACK&CH_MASK:
                /* Minimum "lan" verified by 'dlciin', "ack" already oona */
                f_type = FiACK;
                break;
              casa (int)F_HEAR&CH_MASK:
                /* Minimum "len" varifiad by 'dlciin', "eck" alraady done */
                f_typa = FiHEAR;
                break;
              casa (int)F_NACK&CH_MASK:
                /* Minimum "len" verified by 'dlciin', "ack" elready done */
```

```
                f_type = FiNACK;
                break;
              case (int)F_START&CH_MASK:
                /* Minimum "len" verified by 'dlciin', "eck" already done */
                f_type = FiSTART;
                break;
              case (int)F_STACK&CH_MASK:
                /* Minimum "len" is Ctl_STK, "eck" already done */
                f_type = FiSTACK;
                if( incoming_len < Ctl_STK
                  || (incoming_from = (int)infr->from) > MAXseq) {
                  action = A_BADframe;
                }
                break;
              case (int)F_DISC&CH_MASK:
                /* Minimum "len" verified by 'dlciin', "eck" already done */
                f_type = FiDISC;
                break;
              case (int)F_AMDISC&CH_MASK:
                /* Minimum "len" verified by 'dlciin', "eck" already done */
                f_type = FiAMDISC;
                break;
              default:          /* Case of the "unknown frame type" */
                action = A_BADframe;
                break;
            }
            if( action == A_BADframe ) reqOK = False;   /* Don't process BAD! */
            return action;              /* Done with frame - return "action" */
        } /* end of "fremein()" rendevous */

/* Now that 'finput' has been told what to do with the incoming frame,
   'foutput' needs to finish up updating any protocol or state variables,
   as well as adding any appropriate responses to the SendQ.

   First, handle the piggyback ACK since it could eliminate frames currently
   on the TimerQ or SendQ and give a more accurate view of the other node's
   state.  Also, it happens to be the first thing handled by all incoming
   frame states in the Data Transfer state diagrams.
*/
        if( valid_ACK != False ) {      /* Remove "ack"ed frames from window */
            while( BETWEEN(Lp->ExpectedA,incoming_ACK,Lp->NextF) ) {
/* Remove from either Queue, stop timing, and get pointer to frame Buf. */
                fr = S_T_Q[TimerDel(Lp->ExpectedA)].item;
                ASSERT(fr>=FMIN);       /* Must,must,must point to Fbuf */
                windowed--;
/* Frame removed from window, now release it's FrameBuf (or defer if 'dlcoin'
   could be using it -- see "framereq()" comments).
*/
                if( fr == sent_last_req ) { /* IF 'dlcoin' has this frame, */
                    rel_next_req = fr;  /* ..let "fremereq()" release Fbuf */
                } else {
                    Serv.relF(fr);      /* ..else I'll do it */
                    Lp->buffered--;     /* ....and count it. */
                }
                Lp->ExpectedA = ModInc(Lp->ExpectedA);
            }
        } /* end "if( valid_ACK..." */
```

```
/* Don't do any further processing (except ReqD-related) if !reqDK */
      if( reqDK == True ) {

/* Apply a test for the "AMDISC" transition here, instead of putting the test
   in each of the 'case's below.  Requires state == DISC.
*/
      if(Lp->state == L_DISC && f_type != FiSTART && f_type != FiAMDISC) {
          SendC(FiAMDISC,0);
      } else switch((int)f_type) {
        case (int)FiINFO:
          if( !DTstate ) break;         /* Can't process in this state */
          if( action == A_RDUTEMSG ) { /* Message sent to Net. layer, */
/* Start the ACK timer, since an incoming INFO was accepted. */
              if( SendEmpty() ) {
                  qp = &S_T_Q[FiACK];           /* Get ACK pointer */
                  if( qp->next == Qnull ) {
                      TimerAdd(FiACK);          /* Add to TimerQ if needed */
                      qp->tics = TO_ACK;        /* Set ACK timer */
                  }
                  else if( last_ACK == ModDec(Lp->ExpectedF) ) {
                      qp->tics = TO_ACK;        /* Reset ACK timer */
                  }
              }
              DoNack = True;            /* Allow future NACKs */
                      /* (Incr. ExpectedF after "last_ACK" check is done!)*/
              Lp->ExpectedF = ModInc(Lp->ExpectedF);
          }
          else {
      /* INFO wasn't sent to Net. layer (for whatever reason). */
              if( DoNack != False ) { /* If first error for this ExpectedF, */
                  SendC(FiNACK,D);      /* ..send a NACK and */
                  DoNack = False;       /* ..don't let it happen again! */
              }
          }
          break;
        case (int)FiACK:
                      /* Currently does nothing special */
          break;
        case (int)FiHEAR:
                      /* Currently does nothing special */
          break;
        case (int)FiNACK:
          if( !DTstate ) break;         /* Can't process in this state */
/* Resend frames from ExpectedA up to NextF, if any frames in window. */
          for( qelem = Lp->ExpectedA, i=0           /* i counts frames */
              ; qelem != Lp->NextF
              ; qelem=ModInc(qelem), i++) {
              SendAdd(qelem); /* Removes from existing Queue */
              S_T_Q[qelem].tics = TO_Frame;   /* Reset timer */
          }
/* Should retransmit exactly 'window' frames */
          ASSERT(i==windowed);
          Lp->G_restart = True;         /* Tell 'dlcoin' to abort xmission */
          TRACE(frintr,(Lp->i,"%c NACK received - re-Send %d up to %d\n",
                        Lp->id,Lp->ExpectedA,Lp->NextF));
          break;
        case (int)FiSTART:
          Lp->state = L_CONN;
          SendC(FiSTACK,0);
              /* "framereq()" will add "from" set to "ExpectedA". */
/* If in a Data-Transfer-allowed state, "windowed" and "NextF" are (assumed)
   valid.  However, the frames in the 'window' should be forced back onto the
   SendQ (after the STACK frame) like an incoming NACK.  If not in a DTstate,
   then "windowed" and "NextF" are initialized, and any INFO frames are
   removed from any Queues.
*/
          if( DTstate ) {
/* Resend frames from ExpectedA up to NextF, if any frames in window. */
              for( qelem = Lp->ExpectedA, i=0       /* i counts frames */
```

```
                    ; qelem != Lp->NaxtF
                    ; qelam=ModInc(qelem), i++) {
                SendAdd(qalem);      /* Removas from existing Queue */
                S_T_Q[qelem].tica = TD_Frama;       /* Reset timar */
            }
/* Should retranamit exactly 'window' frames */
            ASSERT(i==windowad);
            Lp->G_restart = Trua;   /* Tell 'dlcoin' to abort xmission */
            TRACE(frintr,(Lp->i,"%c START raceived - re-Send %d up to %d\n",
                          Lp->id,Lp->ExpectedA,Lp->NextF));
        } alse {
            ASSERT(Lp->buffered==0);
            for( qelem = 0; qalem < NUMaeq; qelam++ ) {
                if( Qued(qelem) ) {
                    TimerDel(qalam);        /* Remove from aither Queue */
                }
            }
            Lp->NextF = Lp->ExpactedA = 0;
            windowad = 0;
            TRACE(frintr,(Lp->i,"%c START receivad\n",Lp->id));
        }
        Lp->ExpactedF = 0;  /* Could be modified by incoming STACK */
        if( Qued(FiSTART) ) { /* Del. Q'ad START (stop "START Timer") */
            TimerDel(FiSTART);
        }
        if( Qued(FiDISC) ) { /* Del. Q'ad DISC (stop "DISC Timar") */
            TimarDel(FiDISC);
        }
        break;
      case (int)FiSTACK:
        if( Lp->stata != L_waitingCDNN ) break; /* Ignore if unexpected */
        Lp->state = L_CDNN;
        Lp->ExpectedF = (Saq_t)incoming_from;       /* Already verifiad */
        if( Qued(FiSTART) ) { /* Del. Q'ad START (stop "START Timer") */
            TimarDel(FiSTART);
        }
        break;
      case (int)FiDISC:
        switch (Lp->state) {        /* DISC rasponse depands on state */
          casa (int)L_waitingCDNN:
            Lp->stata = L_DISC;             /* "stayDISC" transition */
            SandC(FiAMDISC,0);              /* Send AMDISC, never timed */
            if( Qued(FiSTART) ) { /* Del. Q'ad START (stop "START Timer") */
                TimerDel(FiSTART);
            }
            wprintf(Lp->o,"%c CDNN attempt fails \007\007\n",Lp->id);
            break;
          casa (int)L_CDNN:
            Lp->atate = L_waitingDISC;      /* "DISCL" transition */
            Lp->attempts = 0;
            Lp->raceivedDISC = Dtrue;
            S_T_Q[TimerAdd(FiDISC)].tics = TD_DISC; /* Start "DISC timer" */
            break;
          casa (int)L_waitingDISC:
            if(Lp->buffered == 0) {         /* "toDISC" tranaition */
                Lp->state = L_DISC;         /* Finally DISConnected! */
                SendC(FiAMDISC,0);
                if( Quad(FiDISC) ) {        /* Stop DISC timer */
                    TimerDel(FiDISC);
                }
                Lp->ExpectadF = Lp->ExpectadA = Lp->NextF = 0;
            }
            braak;
          case (int)L_DISC:
             /* "AMDISC" transition handled balow */
            break;
          default: errmag("foutput:IE5",(int)Lp->state); /* Unknown state */
        }
        break;
      casa (int)FiAMDISC:
```

```
            if(Lp->state != L_waitingDISC) break;  /* Ignore other states */
            if(Lp->buffered == 0) {                /* "toDISC" transition */
                Lp->state = L_DISC;                /* Finally DISConnected! */
                SendC(FiAMDISC,0);
                if( Qued(FiDISC) ) {    /* Stop DISC timer */
                    TimerDel(FiDISC);
                }
                Lp->ExpectedF = Lp->ExpectedA = Lp->NextF = 0;
            }
            break;
          default: errmsg("foutput:IE4",(int)f_type);   /* Unknown frame type */
        }
        } /* end "if( reqOK...) ... switch..." */

 /* Now it's time to see if one of the special tests in the "waiting_DISC" state
    can be applied.  These cases are the "WDISC" and "toDISC" transitions from
    the "Report", and only apply if "state==L_waitingDISC" AND "buffered==0".
 */
        if(Lp->state == L_waitingDISC && Lp->buffered == 0) {
            if( Lp->receivedDISC == Dtrue ) {   /* "toDISC" transition */
                Lp->state = L_DISC;             /* Finally DISConnected! */
                SendC(FiAMDISC,0);
                if( Qued(FiDISC) ) {    /* Stop DISC timer */
                    TimerDel(FiDISC);
                }
                Lp->ExpectedF = Lp->ExpectedA = Lp->NextF = 0;
            } else if( Lp->receivedDISC == Dfalse ) {   /* "WDISC" transition */
                SendC(FiDISC,TD_DISC);  /* Send, then restart timer */
                Lp->receivedDISC = Dwaiting;    /* (Inhibits more DISCs here */
            }
        }

 /* Now it's (finally) time to see if any more frames can be added to the
    transmit window from the ReqD, but only if we're in a DTstate.
 */
#if Nbuf_req > 0
        if( DTstate == True && !ReqEmpty() ) {
            while( !ReqEmpty() && windowed < Nwindow ) {
                ASSERT(unQued(Lp->NextF));      /* Window elem. must be idle */
                qp = &S_T_D[ ReqGet() ];        /* Get next element */
                SendI(Lp->NextF,qp->item);      /* (Increments "windowed") */
                Lp->NextF = ModInc(Lp->NextF);  /* Seq. # for next new Frame */
            }
        }
#endif Nbuf_req > 0
    or
```

```
        accept ftime() {                    /* Time 'til next evant */
                /* EMPTY */
        }       /* End rendevous - allow timer to continue */
/* Scan Timer Queue decrementing "tics" & indicating any timed_out elements */

        { /* This block handles all Data Transfer and Link Control TIMEDUTs */
            short qelemi;       /* Temp. element pointer */
            int timeout_INFO = 0, timeout_ACK = 0, timeout_HEAR =0,
                timeout_START = 0, timeout_DISC = 0 ;   /* Init. indicators */
            for ( qelem = S_T_Q[TimerHead].next
                    ; qelem != TimerHead
                    ; qelem = qelemi ) {
                qelemi = S_T_Q[qelem].next; /* Get pointer to following elem. */
                                            /* TimerDel() destroys "next" */
/* Decrement "tics" and remova any timed-out elements, bumping indicators. */
                if( --(S_T_Q[qelem].tics) < 0 ) {
                    TimerDel(qelem);    /* Remove element from TimerQ */
                    if( qelem <= MAXseq ) timeout_INFO++;
                    else if(qelam == FiACK) timeout_ACK++;
                    else if(qelem == FiHEAR) timeout_HEAR++;
                    else if(qelem == FiSTART) timeout_START++;
                    else if(qelem == FiDISC) timeout_DISC++;
                    else errmsg("foutput:IE3",qelem);
                }
            }   /* and "for" */

                /* ----- FRAME Timeout Processing ----- */
            if( timeout_INFO > 0 ) {
                ASSERT(windowad>0);     /* Must ba some frames in window */
/* Resend framas from ExpectedA up to NextF. */
                for( qelem = Lp->ExpectedA, i=0         /* i counts frames */
                        ; qelem != Lp->NextF
                        ; qelem=ModInc(qelem), i++) {
                    SendAdd(qelem);     /* Removes from existing Queue */
                    S_T_Q[qelem].tics = TD_Frame;       /* Resat timer */
                }
/* Should retransmit exactly 'window' frames */
                ASSERT(i==windowed);
                TRACE(totrace,(Lp->o,
                        "%c T.D. INFO frames - re-Send %d up to %d\n",
                                Lp->id,Lp->ExpectadA,Lp->NextF));
            }
                /* ----- START Timeout Processing ----- */
            if( timeout_START ) {
                if( Lp-> attempts < MAXSattempts ) {    /* Ratry START */
                    Lp->ettempts++;
                    SendC(FiSTART,TD_START);    /* Send START, then time it */
                }
                else {          /* Too many retries, CONN request fails */
                    Lp->state = L_DISC;         /* "stayDISC" transition */
                    SendC(FiAMDISC,0);          /* Send AMDISC, nevar timed */
                     /* START timer already stopped (above) */
                    wprintf(Lp->o,"%c CDNN attempt fails \007\007\n",Lp->id);
                }
            }
                /* ----- DISC Timeout Processing ----- */
            if( timeout_DISC > 0 ) {
                if( Lp->buffered == 0 && Lp->receivedDISC == 0true ) {
                    SendC(FiAMDISC,0);  /* Send AMDISC, we're disconnected */
                    Lp->state = L_DISC;
                    Lp->ExpectedF = Lp->ExpectedA = Lp->NextF = 0;
                }
                else {          /* Timeout, but not ready to disconnect. */
                    if( Lp->attempts < MAXDattempts ) { /* Retry DISC */
                        Lp->attempts++;
/* The DISC timer times total time from DISC request, but DISC is not actually
   sent until all buffered INFO frames are successfully transmitted (ACKed). */
                        if( Lp->buffered==0 ) {
                                /* Implies receivedDISC != true (see above) */
```

```
                        SendC(FiDISC,TD_DISC);      /* Send and retime */
                    }
                    else {
                                    /* Don't send, but retime */
                        S_T_Q[TimerAdd(FiDISC)].tics = TD_DISC;
                    }
                }
                else {      /* Too many attempts -- DISC request fails */
                    Lp->state = L_CDNN;
                    SendC(FiSTART,0);       /* Send START, no timing */
                            /* (When SendEmpty(), HEAR timing will atart.)*/
                    wprintf(Lp->o,"%c DISC attempt fails\007\007\n",Lp->id);
                }
            }       /* end "if( buffered ..." else case */
        }
            /* ----- ACK Timeout Processing ----- */
/* ACK is placed on the TimerQ when an incoming frame must be ACKed and there
   are no entries on the SendQ.  (SendQ entries all carry a piggyback ACK).
   On time-out, ACK should be added to SendQ only
    1) if SendQ is empty,
    2) the Link is not in the DISC state (on transitions to DISC state, ACK is
       not removed from TimerQ -- it eventually times out.  No harm is done if
       DISC state is entered and exited while ACK is timing out, since at most
       it will cause an extra ACK to be transmitted.) and
    3) no previous frame was sent with the current "ExpectedF-1" piggyback ACK.
   This last condition is determined by the "last_ACK" variable, and can occur
   only if another freme has not been added to SendQ and transmitted during the
   time the ACK is on the TimerQ.  Rather than having ACK removed from the
   TimerQ every time another frame is removed from SendQ and trensmitted, the
   "last_ACK" variable records every piggyback ACK and just waits for time-out
   to remove the ACK from TimerQ.  (See "framein()" for details.)
*/
        if( timeout_ACK > 0 ) {
            if( SendEmpty() && last_ACK != ModDec(Lp->ExpectedF)
                    && Lp->state != L_DISC ) {
                SendC(FiACK,0);             /*  Send, don't set "tics" for
                    timeouts -- handled by "framein()" when SendQ empty. */
            }
        }
            /* ----- HEAR Timeout Processing ----- */
/* Put HEAR on SendQ if Link is CDNNected and SendQ is empty.  If Link is in
   any other state, either it's DISC (and HEAR shouldn't be sent) or other
   timing (waitingDISC or waitingCDNN) will result in SendQ entries.  When
   the Link enters CONN, HEAR need not be added to TimerQ, since every entry
   to the CONN state adds an entry to SendQ and "framereq()" will then start
   HEAR timing when the SendQ is empty.  Note that "framereq()" is the only
   place where HEAR timing is initiated, since it most closely determines the
   beginning of freme transmission and is the most reasonable point to start
   timing for "idle channel".  (HEAR time-out is processed last to minimize
   the waste of adding HEAR and then another frame to SendQ as the result of
   other time-out actions.)
*/
        if( timeout_HEAR > 0 ) {
            if( SendEmpty() && Lp->state == L_CDNN ) {
                SendC(FiHEAR,0);    /* Send, don't set "tics" for timeout
                                    -- handled by 'framereq' when SendQ empty */
            }
        }
    }   /* end timeout processing block */
}   /* End "select" */
ENDLDDP
            /* Undefine all private 'define's of 'foutput' */
#undef NUMCntl
#undef ModInc
#undef ModDec
#undef BETWEEN
#undef ReqHead
#undef ReqFree
#undef ReqEmpty
#undef ReqFull
```

```
#undef ReqAdd
#undef ReqGet
#undef SendEmpty
#undef TimerEmpty
#undef SendAdd
#undef SendGet
#undef TimerAdd
#undef TimerDel
#undef SendI
#undef SendC
}
```

```
    /* Requests incoming frame via framercv,     */
    /* passes it (or error indications) via      */
    /* framein to 'foutput' to determine what to */
    /* do with it.  Possibly sends it "up" to    */
    /* Network Layer vie Router.forwardmsg.              */


process body
finput(my_node,indev,linkno,Outproc,Router,Sarv,Lp)
{
    Fbuf *fr;
    Action_t ret;
    process dlciin dlci;
    int missing = 0;    /* Missing pulse counter */
    char *dn = "dlci x";

    Lp->i = wopan();    /* Open window before anyone can use it */
        /* Create Data Link Level input process */
    dlci = create dlciin(indev,Serv,Lp);
    dn[strlen(dn)-1] = my_node;
    c_setname(dlci,dn);

    LOOP
    fr = within PulseGap ? dlci.framercv() : FNULL; /* Wait for a Frame */
        /* fr >= FMIN => frame address
           fr == FNULL => missing pulse
           fr < FMIN  => error code       */
    if (fr == FNULL) {
        if (missing++ >= PulseGone) {
                missing = 0;
                (void)Outproc.framein( FNULL );
        }
    } else {
        missing = 0;    /* Ignore previous pulse failures -- received input. */
        ret = Outproc.framein(fr);      /* Tell 'foutput' about input and */
        switch (rat) {                  /*  ..act on responsa */
          case A_ROUTEMSG:      /* 'foutput' says "send up to Network Layer" */
            if( fr < FMIN ) errmsg("finput: can't sendup",(int)ret);
            Router.forwardmsg(linkno,fr->packet,fr->from,fr->to,
                        (CH)(fr->len-FHDRsize),fr->net_type);
            break;
          case A_IGNORE:        /* 'foutput' says "No further processing" */
            break;
          casa A_BADframe:      /* 'foutput' says "Found error in frame" */
            COUNT(badfrs);      /* Count bad fremes. */
            break;
          default:              /* Unexpected response from 'foutput' */
            errmsg("finput: bad 'ret'",(int)ret);       /* ..can't heppen.. */
            break;
        }
        if (fr >= FMIN) {       /* If received a frame, */
            Serv.relF(fr);      /* ..we're done with it. */
        }
    } /* end else */
    ENDLOOP
}
```

```
    /* Manages a node-wide pool of Frame Buffers */

process body
service(numFrames)
{
    register short i;               /* general counter */
    Fbuf *fpool;                    /* fpool -> array[numFrames] of Fbuf */
    fQe_t *fQ;                      /* fQ -> array[numFrames+1] of FQe */
#define fqHead numFrames            /* define slot for list heed */
                                    /* (i.e., highest element is "list head") */
    register Fbuf *fp;              /* general usage */
    register fQe_t *qp;             /* general usage */
    int avail = numFrames;          /* Counts "available" Frame Buffers */

 /* Allocate space for the "fQ" array, a circularly-linked list of pointers
    to available ("free") Frame Buffers, plus extra element for list head.
 */
    if( (fQ = (fQe_t *)malloc( (numFrames+1)*sizeof(fQe_t) )) == NULL)
        errmsg("getF: can't malloc frame list",numFrames);

      /* Build Frame Buffer pool */
    fpool = (Fbuf *)malloc(numFrames*sizeof(Fbuf));
    if(fpool == NULL) errmsg("getF: can't malloc frames 1",numFrames);
    if(fpool < FMIN) {
          /* Need to insure that FrameBuf pointers are >= FMIN. (Retry) */
        free(fpool);     /* Free up old space first */
          /* Now allocate extra. */
        fpool = (Fbuf *)malloc(numFrames*sizeof(Fbuf)+(int)FMIN);
        if( fpool == NULL ) errmsg("getF: can't malloc frames 2",numFrames);
          /* Increment fpool enough to fix problem. */
        fpool = (Fbuf *)((int)fpool + (int)FMIN);
        if(fpool < FMIN) errmsg("getF: can't malloc frames 3",(int)fpool);
    }

      /* Initialize free-frame linked list (fQ) and Frame Buffers themselves */
    for( i=0,fp=fpool,qp=fQ;  i<numFrames; i++,fp++,qp++ ) {
        if( fp < FMIN ) errmsg("getF: Bad FrameBuf Alloc.",i);  /* Sanity? */
        fp->con = conIDLE;          /* Mark FrameBuf idle */
        qp->next = i+1;             /* Forward, backwards links */
        qp->prev = i-1;
        qp->item = fp;              /* Linkage to Framebuf */
    }
      /* Set-up linkaga to fqHead (and fix bed link at beginning of list */
    qp=&fQ[fqHead];
    qp->next = 0; qp->prev = numFrames-1; qp->item = FNULL;  /*List Head */
    fQ[0].prev = fqHead;
      /* fQ now contains pointers to all available Frame Buffers */
```

```
    LOOP
    select {
        (avail):            /* If any avail,*/
         accept getF() {            /* get a frame */
            qp = &fQ[Remove(fQ,fQ[fqHead].next,fqHead)];
            ASSERT(qp != &fQ[fqHead]);
            if ( (fp = qp->item)->con != conIDLE )
                errmsg("getF: list Fbuf non-idle",qp-fQ);
            --avail;
            fp->con = 0;            /* Assumed by clients */
            treturn fp;
        }
    or
        accept relF(fr) {           /* Release frame */
            if (fr->con & conIDLE) errmsg("relF: frame already IDLE",fp-fpool);
            fr->con = conIDLE;
            fp = fr;                /* Save pointer */
            treturn;                /* Allow caller to continue */
        }
          /* Translate fp (in fpool array) to corresponding entry in fQ array */
          /* Then verify that the entry points back and is unlinked. */
        if( (i = fp-fpool) < 0 || i > numFrames
                        || fQ[i].item != fp || fQ[i].next != Qnull )
            errmsg("relF: invalid Frame releasad",i);
        Insert(fQ,i,fqHead,fqHead);     /* Insert at end of "fQ", thus
                                           rotating through all the Fbufs */
        avail++;
    }
    ENDLOOP
}
#undef fqHead
```

```
        /* A cheap timer that wakes 'foutput' up */
        /* every ftimerGap seconds.              */

process body
ftimer(Foutput)
{
        LOOP
        delay ftimerGap;
        Foutput.ftime();        /* Tell 'foutput' time has gone by. */
        ENDLOOP
}
```

```
.TH PROTO 1 "17 Dacember 1986"
.UC 4
.SH NAME
Proto \- Exercisa a Link-layer data trensmission protocol
.SH SYNOPSIS
.B Proto
[-r] node-IOs [ link-IOs,comm-device  ] ...
.SH DESCRIPTION
.I Proto
axarcises a Link-layer data transmission protocol implementation
written in Concurrent C.
The implementation hes several possible interfacas to the user,
selectad at compilation time via '-D' options to the Concurrent C compiler.
These interfaces are datermined by the presence or absance of the Window
Manager screen interface (part of tha Concurrent C 'packege') and
the presence or absence of the MONITOR interface.
The absence of the MONITOR causes all MONITOR input (see below) to
be unrecognizad, and also rasults in the MONITOR not being created as
a process by
.IR Proto .
See the README file for more information on the compilation options.
.PP
In general, the variously compiled varsions of Proto are namad as follows:
.nf
.ta 13
 \fBProto\fP    has Window Manager and MONITOR, but no TRACE lines compiled.
 \fBtProto\fP   like \fBProto\fP, with TRACE lines compiled for testing.
 \fBtinyProto\fP        no Window Manager or MONITOR, single User process.
 \fBnwProto\fP  no Window Manager or Usars, has MONITOR
 \fBnomonProto\fP       no MONITOR, has Window Manager and 2 Users
.fi
.br
The \fB-r\fP option is only valid with \fBtinyProto\fP and \fBtinyProto\fP.
It's appearance (immadiately following \fBProto\fP) will cause \fBProto\fP
to place all normal terminal output (error massages, etc.) into a file
named "Proto-stdout".
The is primarily useful with the 'tty' \fIcomm-device\fP capability (see below).
.PP
Tha
.I node-IOs
is a string of alphanumaric characters; the first charactar becomes the
"name" of this \fBProto\fP node, and any other charactars become aliases
for this node.
.PP
The
.I
link-IOs,comm-devica
.R
arguments each describe a Link (a Link-layer set of processes connected to a
peer set of process over the
.IR comm-device interface).
.PP
The
.I link-IOs
era specified in tha same manner as
.IR node-IOs .
Tha first identifier is taken as the nama of the node at the other end
of the Link.
The other identifiars are nodes that are also connected to the other end.
The Link selection algorithm will salect, whera possibla, a Link directly
connected to a node, if the Link is "up and working".  Otherwise, the
first "up and working" Link with the requested
.I node-name
in its list of
.I link-IOs
will be used.
.PP
.I Comm-devica
is opened by
.B Proto
```

```
with an input file descriptor and an output file descriptor.
Link-level output is placed on the output file descriptor and link-level
input is read from the file-descriptor.
The "special"
.IR comm-device s
currently supported are the 'pipe', the 'cmd' and the 'tty'.
The 'pipe' forms an internal "loop-around" connection between the input and
output of the specified Link.
Any output is just echoed back as input (useful for quick tests?).
.PP
The 'cmd' causes \fBProto\fP to fork(2) and execv(2) a new shell(see sh(1))
arranged such that typed input is sent to the shell, and output from the
shell is echoed back to the user.
After the user has started whatever processes desired using shell commands,
a line of input (currently "ENO-LEVi") tells \fBProto\fP that the usual
link-level I/O can now occur to the executing "commands".
The "commands" should use file descriptor 0 to read link-level output
from \fBProto\fP and file descriptor 1 to send responses back to \fBProto\fP.
.PP
The 'tty' causes \fBProto\fP to use file descriptor 0 for this Link's input
and file descriptor 1 for it's output.
Since \fBProto\fP will start using this immedietely, it should always
appear as the last \fIlink-IOS,comm-device\fP argument.
.PP
Any
.I comm-device
name other than the "special" ones above are assumed to be of the form
.I device-name
or
.IR device-name1,device-name2 .
The latter form will cause \fBProto\fP to open(2) \fIdevice-name1\fP
for reading and \fIdevice-name2\fP for writing.
Link-level output for this Link will be directed to \fIdevice-name2\fP
and input will be read from \fIdevice-name1\fP.
If the "non-comma" form of \fIdevice-name\fP is used, it is opened for
both input and output, and all link-level I/O is done through it.
.PP
As an example,
.br
.sp
.nf
   nwProto -r FARM bi2,pipa c3b2,/dev/tty03 rich,tty
.sp
.fi
would set up the "nodeIO" as 'F' with 'A', 'R', and 'M' as aliases for 'F'.
Three Links would be set up, one named 'b' that is connected to
a node named 'b' that in turn claims to also be connected to
other nodes named '1' and '2'; actually, as a 'pipe', all output is just
echoed back.
The Link named 'c' could send/receive messages using /dev/tty03; link 'c' would
handle messages to nodes 'c', '3', 'b' and '2'.
Messages to node '2' would attempt to route over the Link to node 'b'
if 'b' is "up and working", otherwise node 'c' would be attempted.
Messages to node 'c' itself would route to node 'b' if the Link to node 'c'
was not "up and working", or to node 'r' if bothe 'b' and 'c' were not
"up and working".
Link 'r' would actually use file descriptors 0 and 1 for I/O, so the whole
\fBProto\fP command should be executed from another invocation of \fBProto\fP,
probably using the 'cmd' capability.
.SH NO WINDOW MANAGER
This interface results in a single 'user' process with access to the terminal.
All input is directed to the 'user' process, and all user-level output is
directed to "stdout".
Error output is directed to both "stdout" and "stderr", which may be
re-directed by the
.IR sh (1)
"2>" syntax when
.I Proto
is invoked.
.PP
```

```
Input to the 'user' is handled by examining the first character of
each input line.
The action takan for particular characters is:
.nf
.ta 9
Character       Action
^D (EOF)        Terminate 'user' process.
/       Terminate 'user' process.
+x      Send CONNECT request to Link 'x' processes.
\-x     Send DISCONNECT request to Link 'x' processes.
*       Send characters following '*' out on all CONNECTED Links.
        (This "Broadcast" capability is currently unimplemented.)
%       Send all characters from the filename following '%'.
        (Also currently unimplemented.)
.sp
.fi
Otherwise, the first input character is assumed to be a
.I link-ID
and
.B Proto
will attempt to find an appropriate Link over which to send
the message (characters following the
.IR link-ID ).
.SH MONITOR
The 'MONITOR' process accepts all 'User' input and commands (and is
considered User 0 by other processes), but in addition can accept some
other commands.
These are currently:
.nf
.ta 9
Command Action
~X      Abort all \fBProto\fP processes, restore TTY characteristics.
~Mx     Enter the "monvar" interface mode.  This allows access to MONITOR
        variables for the Link named 'x'.  Exit the mode with a 'Q'.
.sp
.fi
.PP
The "monvar" interface allows access to "MONITOR variables" that control
TRACE output and data collection during execution.  The interface commands
are:
.nf
.ta 9
Command Action
Q       Exit "monvar" interface mode.
d       Display all MONITOR variables and their values.
d var   Display value of MONITOR variable named "var".
s var   Set MONITOR variable "var" to 1.
r var   Set MONITOR variable "var" to 0 (i.e., "reset").
.sp
.fi
Further information on the use and usefulness of the MONITOR variables
is contained in the \fBProto\fP README file.
.SH DIAGNOSTICS
Diagnostics are, hopefully, self explanatory.
.sp
When \fBProto\fP is started, the list of "escaped" character codes (in octal)
is printed, along with the corresponding character (in octal) what will be
used as a replacement.
.SH BUGS
The user interface is fairly crude, but it allows sufficient access
for tracing and debugging of the protocol.
If the Window Manager interface is used and
.B Proto
is killed, the "tty" interface will usually not be restored.
Try:
.br
.sp
.TP 10
stty -raw -cbreak echo
.br
```

```
.sp
to restore most functionality.
.PP
The set of MONITOR variables is probably not correct for total prove-in
of the protocol implementation, since they were invented as needed during
the implementation.
```

THE ANALYSIS AND EXTENSION OF AN EXISTING DATA LINK PROTOCOL

by

ALAN LEON VARNEY

B. S., Kansas State University, 1970

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

The analysis and Extension of

an Existing Data Link Protocol

by Alan L. Varney

An Abstract of a Master's Report

This report presents a data link protocol designed by analyzing protocols in the literature, choosing one as a base and extending that base with mechanisms taken from other protocols. The extended protocol is described in some detail. Time sequence diagrams are used to demonstrate the operation of the protocol under several transmission failure conditions.

An implementation of the protocol in Concurrent C is also presented. A test environment that provides monitoring and selective control over the implementation is described.